НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:

Завідувач кафедри

_____ С. Г. СТІРЕНКО

«___»_____20__ р.

Дипломний проект

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інженерія програмного забезпечення комп'ютерних систем»

спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Система пошуку маршрутів в топологіях на основі еволюційних алгоритмів»

Виконав (-ла):
студент IV курсу, групи ІП-64
Морозов Артур Сергійович _____

Керівник:
доцент, к.т.н.
Волокита Артем Миколайович _____

Консультант з нормо контролю:
Професор, доктор технічних наук
Сімоненко Валерій Павлович _____

Рецензент:

_____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2020 року

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМ. ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення комп'ютерних систем»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ *Сергій СТІРЕНКО*
(підпис)
"___"_____20__ р.

ЗАВДАННЯ

на дипломний проект студенту

Морозову Артуру Сергійовичу

1. Тема проєкту «*Система пошуку маршрутів в топологіях на основі еволюційних алгоритмів»*

керівник проєкту Волокита Артем Миколайович, доцент, к.т.н., затверджені наказом по університету від *«07»* травня __2020р. №__1081-с__

2. Термін здачі студентом закінченого роботи__*27 травня 2020р.*
3. Вихідні дані до проекту*: технічна документація, теоретичні та статистичні дані.*
4. Зміст пояснювальної записки:

   Розділ 1. Огляд топологій та евристичних алгоритмів

   Розділ 2. Вибір використовуваних технологій

   Розділ 3. Деталі розробки системи

Розділ 4. Аналіз розробленої системи

5. Консультант роботи, з вказівкою розділів роботи, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Розділ 1	доцент, к.т.н. Волокита А. М.		
Розділ 2	доцент, к.т.н. Волокита А. М.		
Розділ 3	доцент, к.т.н. Волокита А. М.		
Розділ 4	доцент, к.т.н. Волокита А. М.		

6. Дата видачі завдання *14.12 .2019 року*

КАЛЕНДАРНИЙ ПЛАН

п/п	Найменування етапів дипломного проекту (роботи)	Строк виконання етапів проекту(роботи)	Примітки
1.	Затвердження теми роботи	14.12.2019-19.12.2019	Виконано
2.	Вивчення та аналіз завдання	19.12.2019-19.03.2020	Виконано
3.	Розробка архітектури та загальної структури системи	19.03.2020-30.03.2020	Виконано
4.	Розробка структур окремих підсистем	30.03.2020-10.04.2020	Виконано
5.	Програмна реалізація системи	10.04.2020-20.04.2020	Виконано
6.	Оформлення пояснювальної записки	15.04.2020-10.05.2020	Виконано
7.	Захист програмного продукту	25.04.2020	Виконано
8.	Перед захист	28.05.2020	Виконано
9.	Захист	19.06.2020	Виконано

Студент Артур МОРОЗОВ _____

(підпис)

Керівник Артем ВОЛОКИТА _____

(підпис)

АНОТАЦІЯ

У даній дипломній роботі була розроблена система для пошуку маршрутів у топологіях, використовуючи еволюційний алгоритм. Дана система дозволяє користувачам за допомогою взаємодії із нею знаходити маршрути у повнозв’язних топологіях.

Дана система дозволяє обчислювати маршрути декількома способами та конфігурувати їх власноруч для користувача. Першим конфігураційним параметром для обчислень, виконуваниз системою, є адреса, за якою звернувся користувач до API даної системи. Залежно від цього система може обчислити маршрут одним із декількох алгоритмів, залежно від інших побажань користувача, або за допомогою використання певного конкретного алгоритму.

Також дана система дозволяє конфігурувати побажання користувача щодо обчислень, наприклад, щодо їх часу – від того, чи критичним є час для користувача, залежить спосіб обчислень, і, відповідно, й час, а також специфічні параметри для алгоритмів.

Ключові слова: пошук маршрутів, топології, Java, повнозв’язна топологія.

ANNOTATION

In this diploma work, a system for finding routes in topologies using an evolutionary algorithm was developed. This system allows users to interact with it to find routes in fully connected topologies.

This system allows its users to calculate routes in several ways and configure them by themselves. The first configuration parameter for calculations performed by the system is the URL at which the user accessed the API of the system. Depending on this, the system can calculate the route by one of several algorithms, depending on other user preferences, or by using a specific algorithm.

This system also allows its users to configure the user's wishes for calculations, for example, for their time - whether the time is critical for the user, depends on the method of calculation, and, accordingly, time, and also specific algorithm parameters.

Key words: route search, topologies, Java, fully-connected topology.

ВІДОМІСТЬ ДИПЛОМНОГО ПРОЄКТУ

з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
	А4		Завдання на дипломний проект		
	А4	ІАЛЦ.467200.001 ВП	Відомість проекту	1	
	А4	ІАЛЦ.467200.002 ТЗ	Технічне завдання	3	
	А4	ІАЛЦ.467200.003 ПЗ	Пояснювальна записка	91	
	А3	ІАЛЦ.467200.004 Д1	Структурна схема системи	1	
	А3	ІАЛЦ.467200.005 Д2	Функціональна схема (діаграма класів)	1	
	А3	ІАЛЦ.467200.006 Д3	Алгоритм дій програмного забезпечення	1	
	А4	ІАЛЦ.467200.007 Д4	Текст програмного коду	13	

					ІАЛЦ.467200.001 ВП
Зм.	Арк.	№ документна	Підпис	Дата	
Розробив		Морозов А.С.			Система пошуку маршрутів в топологіях на основі еволюційних алгоритмів Відомість дипломного проекту
Перевірив		Волокита А.М.			
Реценз.					
Н. Контр.		Сімоненко В.П.			
Затверд.					

Літ.	Аркуш	Аркушів
	1	1

НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-64

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЕКТУ

на тему: «*Система пошуку маршрутів в топологіях на основі еволюційних алгоритмів*»

Київ – 2020

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку системи пошуку маршрутів в топологіях на основі еволюційних алгоритмів, а також на подальшу підтримку та вдосконалення розробленої системи.

Областю застосування цієї системи є інтернет речей, суперкомп'ютери, вбудовані пристрої, а також загалом фронтендні клієнти, для яких поставлена задача із знаходження маршрутів у топологіях.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом "Інформатики та обчислювальної техніки" кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка системи пошуку маршрутів у топологіях із використанням еволюційних алгоритмів, що дозволить ефективно вирішувати дану проблему спираючись на особливості клієнта та його побажання щодо обчислень.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є офіційні документації, публікації та статті в мережі Інтернет на дану тему, науково-технічна література.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до програмного продукту, що розробляється

Розроблена система має виконувати такі вимоги:

1) Надати зручні API-ендпоїнти для взаємодії користувача із системою пошуку маршрутів.
2) Надати можливість користувачам передавати вхідні дані для пошуку маршрутів та отримувати у відповідь результат.

3) Надати можливість користувачам вибирати важливі для них параметри, як наприклад якнайменший час обчислення, при відправленні запиту.
4) Надати можливість користувачам власноруч конфігурувати специфічні параметри обчислення для деяких алгоритмів.
5) Надати можливість користувачам власноруч вибирати бажаний алгоритм для обчислення.
6) Надати вичерпну та зрозумілу документацію для API.

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux;

- Java Development Kit версії 9 або вище;

- Intellij IDEA версії 2019.3.4 або вище;

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core (TM) i3-2100T;

- ROM не менше ніж 64 ГБ;

- RAM не менше ніж 4 ГБ;

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	
Вивчення та аналіз завдання	
Розробка архітектури та загальної структури системи	
Розробка структур окремих частин системи	
Програмна реалізація системи	
Виправлення помилок	
Оформлення пояснювальної записки	
Передзахист	
Захист	

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система пошуку маршрутів в топологіях на основі еволюційних алгоритмів*»

Київ – 2020

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

IoT – Internet of Things, концепція обчислювальних мереж із фізичних предметів, що є обладнаними спеціальними технологіями для забезпечення хз взаємодії один із одним.

ПК – Персональний Комп'ютер, електронно-обчислювальний прилад, що призначений для обробки та зберігання інформації.

ANSI C – American National Standards Institute C, це стандарт, що прийшов із мови C, його суть у тому, що слідуючи йому, можна писати легко портуємі програми.

POSIX – Portable Operating System Interface for uniX – це набір певних стандартів, які описують зручний інтерфейс для взаємодією між операційною системою та застосунками під неї.

GUI – Graphical User Interface, це певна система для взаємодії користувача із системою, яка основана на сукупності компонентів у вигляді графічної інформації.

J2EE – Java 2 Enterprise Edition, це обчислювальна корпоративна платформа Java.

MVC – Model-View-Controller, патерн програмування, що базується на виділенні трьох основних слоїв у додатку за їх призначенням.

IoC – Inversion of Control, принцип для зменшення заціплення у програмі.

HTTP – HyperText Transfer Protocol – протокол передачі даних у мережі.

API - Application Programming Interface, набір компонентів однієї сутності для взаємодії із іншими.

DTO – Data Transfer Object, шаблон проектування для передачі даних.

REST – Representational State transfer, архітектурний стиль взаємодії компонентів додатку у мережі.

JSON – JavaScript Object Notation - текстовий формат для обміну даними.

POJO – Plain Old Java Object, компонент, у якому є тільки дані.

ВСТУП

На сьогоднішній день використання комп'ютерних мереж та персональних комп'ютерів все більше набуває популярності майже в усіх напрямках людської діяльності. Таким чином, душе швидко зростає й кількість та складність обчислювальних мереж, як старих так і тільки створених. При цьому зазвичай обчислювальні мережі будуються за архітектурою однієї або декількох мережевих топологій. Варто також зазначити, що у кожній обчислювальній мережі, що складається із принаймні декількох компонентів, постає питання щодо обміну певною інформацією між ними. Продовжуючи цю тему, можна сказати, що дуже часто у таких мережах є критичним час, за який будуть проведені подібні дії.

Окремою ланкою у задачі пошуку маршрутів у різних топологіях є задача комівояжера. Ця проблема є оптимізаційною та типовою. Полягає вона у тому, що необхідним є знайти оптимальний шлях руху для так званого комівояжера, який хоче навідатися до усіх об'єктів, що вказані в умові, і при цьому це необхідно зробити як за найкоротший час, так і витративши як можна менше ресурсів. Як правило, під ресурсами мається на увазі шлях, що потрібно пройти, тобто, він має бути як можна коротший. Також можна зазначити, що задача комівояжера часто та широко застосовується при розробці різних типів програмного забезпечення.

Тому проблема пошуку маршрутів в топологіях є дуже актуальною. Система, що зможе виконувати це завдання, може використовуватися у багатьох місцях людської діяльності. Однією із таких областей є суперкомп'ютери, адже під капотом в них використовуються дуже складні топології, проблема пошуку різних маршрутів у яких є дуже актуальною. Іншою важливою областю застосування маршрутизації в топологіях є інтернет речей (IoT). У даній області усі приймаючі участь речі об'єднуються, як правило, в певну топологію. У свою чергу, знайдення маршрута для передачі

даних за максимально малий відрізок часу по цій мережі є надзвичайно важливим її завданням. Також така система може знайти своє використання у області вбудованих пристроїв (embedded devices), де проблема пошуку маршрутів у різних топологіях, і при цьому й за як можна коротший час є дуже актуальною.

РОЗДІЛ 1
ОГЛЯД ТОПОЛОГІЙ ТА ЕВРИСТИЧНИХ АЛГОРИТМІВ

1.1 Огляд існуючих топологій

Існують різні типи топологій, серед яких можна виділити такі типи, як гіперкуб, зірка, дерево, кільце, решітка, повнозв'язна та лінійна топології.

1.1.1 Гіперкуб

Гіперкуб – одна із найвикористовуваніших і, за сумісництвом, найпростіших топологій. Дана топологія є окремим випадком решітної (mesh) топології.

Гіперкуб відноситься до статичних топологій, тобто, у ній можливий лише один прямий шлях між двома вузлами, що є також фіксованим. Також це означає, що топологія у процесі виконання задачі не змінюватиметься.

Варто також зазначити основні характеристики топології гіперкуб :

1. Діаметр : m
2. Кількість зв'язків : $\frac{\mathrm{mN}}{2}$
3. Зв'язність : $m - 1$
4. Розмір мережі : $m(N - 2^m)$;
5. Порядок вузла : m
6. Ширина бісекції : m [1]

Дана топологія є досить широко використовуваною у практичних завданнях, наприклад, при з'єднанні паралельних процесорів. Існують різні підвиди гіперкубів :

- Одновимірний гіперкуб – відрізок, що поєднує два вузла (рис. 1.1 - a);
- Двовимірний гіперкуб – це квадрат, що може бути створений чотирма вузлами (рис. 1.1 - b);
- Тривимірний гіперкуб – такий, що складається із восьми вузлів та, на відміну від двох попередніх, є об'ємним (рис. 1.1 - c);

- Чотиривимірний гіперкуб – тесеракт – куб у чотиривимірному просторі (рис. 1.1 - d);
- П’ятивимірний гіперкуб – пентеракт – відповідно, куб у п’ятивимірному просторі (рис. 1.1 - e);
- і т.д.

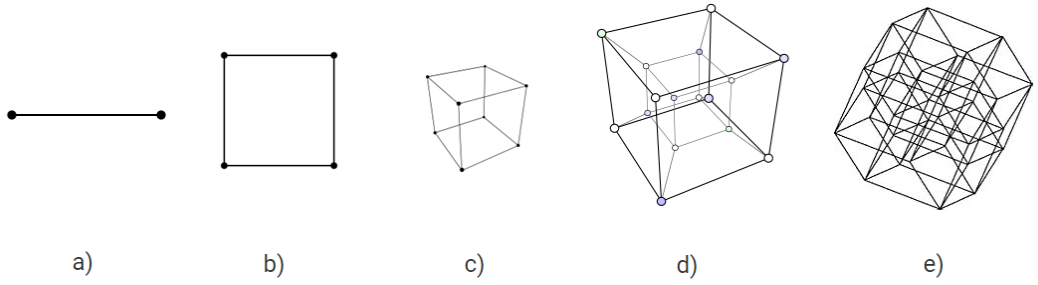


Рис 1.1. Види кубів : a – одновимірний, b – двовимірний, c – тривимірний, d – чотиривимірний, e – п’ятивимірний

Із цієї послідовності можна зробити висновок про алгоритм отримання n – вимірного гіперкуба :

1. Почати із (n – 1) – вимірного куба.
2. Зробити ідентичну копію взятого у п.1 куба.
3. Додати зв’язки між кожним вузлом початкового куба та таким самим вузлом куба-копії.

Гіперкуб розмірності n буде мати кількість вершин 2^n. Збільшення розмірності гіперкуба на одиницю призведе до подвоєння кількості вузлів у ньому, та до збільшення діаметра мережі й порядка вузлів на одиницю.

Маршрутизація у гіперкубі

Обмін повідомленнями, здійснюваний у гіперкубі, базується на двійковому представленні порядкових номерів вузлів. Їх нумерація здійснюється так, щоб для кожної пари суміжних вузлів двійковий варіант представлення номерів даних вузлів відрізнявся тільки по одній позиції. Тобто, наприклад, вузли 0001 та 0011 являються сусідами, тоді як 0001 та 1100 – ні.

Номери вузлів гіперкуба вважаються основою маршрутизації повідомлень у ньому. Можна порахувати, що дані номери в n-вимірному гіперкубі будуть складатись із n кількості бітів, а відправлення повідомлення із одного вузла до іншого здійсниться за n кількість кроків. На кожній ітерації вузол має прийняти рішення, що далі робити із повідомленням – зберегти його та нікуди не відправляти далі, або все ж відправити по одній із можливих ліній. На ітерації під номером і вузол, що на даний момент зберігає повідомлення, має порівняти і-й біт свого двійкового номера із відповідним бітом двійкового номера вузла – призначення. Якщо ці біти виявляються однаковими, пересування повідомлення буде зупинене. Якщо ні, воно має бути передане уздовж відповідної лінії і-го виміру. Лінія і-го виміру - це така, що була сконструйована під час побудови і-вимірного гіперкуба із двох гіперкубів, (і-1)-вимірних. [5]

1.1.2 Зірка

Зірка – це найпростіший тип топології. У даній топології всі наявні вершини є не пов’язаними між собою, проте є пов’язаними з декотрим вузлом, що знаходиться у її центрі. Можна сказати, що зв’язність у цій топології є дійсно мінімальною, а також що всі вершини, окрім центрової, є цілком рівноцінними. Всі вузли у зірці мають перший порядок. Центральний вузол часто також називається концентратором.

Зірка є прикладом двовимірної статичної топології. У процесі виконання її побудова зазвичай не змінюється, а також між двома вузлами у ній є можливим тільки один прямий шлях, що є фіксованим.

Топологія зірка має такі характеристики :

1. Діаметр : 2
2. Кількість зв’язків : $N - 1$
3. Зв’язність : 1
4. Розмір мережі : $N;$

5. Порядок вузла : 1
6. Ширина бісекції : 1 [1]

Існують різні підвиди топології зірка :

- Клешня або лапа – зірка із трьома ребрами (рис. 1.2 - a);
- Зірка із чотирма ребрами (рис. 1.2 - b);
- Зірка із п’ятьма ребрами (рис. 1.2 - c);
- Зірка із шістьмома ребрами (рис. 1.2 - d);
- і т.д.

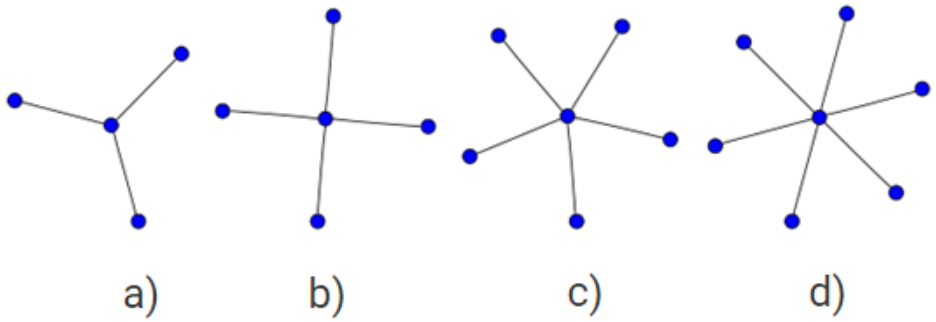


Рис 1.2. Підвиди зірки : a – клешня, b – із чотирма ребрами, c – із п'ятьма ребрами, d – із шістьмома ребрами [1]

Трьохреберні зірки (клешні) також можуть зустрітися при визначенні топологій без клешень, топологій, що не мають підтопологій із клешнями. Також, клешні є окремим винятковим випадком теореми ізоморфізму графів Вітні.

Як відомо, зірка являється особливим підвидом дерева. Тому, як власне й будь-яке з дерев, вона може бути закодована із використанням кода Прюфера. Послідовність Прюфера ж для зірки $K_{1,k}$ буде складатися із $k-1$ кількості копій з головної (центральної) вершини.

Маршрутизація у зірці

У зірковій топології дані, що надходять із передаючої станції мережі, передаватимуться через хаб по всім лініям і до усіх ПК. Таким чином, інформація надійде на усі робочі станції, проте прийметься вона лише тими, які є адресатами. Оскільки передача сигналів у даній топології зірки є

широкомовною, тобто, сигнали, що передаються від ПК, розповсюджуються одночасно по всім напрямкам, то таким чином логічна топології даної локальної мережі буде ідентична логічній шині. [6]

1.1.3 Дерево

Дерево – це ієрархічна топологія. Вона полягає у тому, що вузли вищого рівня у ній є пов'язаними із вузлами низчого рівня. Цей зв'язок встановлюється за допомогою зіркових зв'язків. Таким чином формується комбінація зірок, і тому дану топологію також іноді називають ієрархічною зіркою.

Сама назва топології прийшла до нас із теорії графів. Прийнята назва першого із вузлів у дереві – корінь, наступні вузли, якщо вони знаходяться на вищому рівні, називають батьківськими, а відповідно, вузли низчого рівня – дочірніми вузлами.

Тож кожний дочірній вузол, що при цьому має й зв'язок та утворює ієрархію із вузлами більш низького рівня, буде називатися батьківським вузлом для даних вузлів.

Топологія дерево належить до статичних топологій, оскільки, по-перше, між двома вузлами у ній можливий лише один фіксований шлях, і по-друге, у процесі її використання структура ієрархії зазвичай не змінюється. [1]

Топологія дерево має такі характеристики :

1. Діаметр : $2(log_2 N - 1)$
2. Кількість зв'язків : $\mathrm{N} - 1$
3. Зв'язність : 1
4. Розмір мережі : $N;$
5. Порядок вузла : 3
6. Ширина бісекції : 1 [1]

Щодо класифікації топологій типу дерево, є два основних підвиди дерев:

- Бінарні дерева. Ця топологія, аналогічно до бінарного дерева, має головне правило, яке полягає у тому, що у кожного із батьківських

вузлів може бути не більше, аніж два дочірніх вузла. Бінарні дерева, в свою чергу, можна поділити на два підвиди :

a) Повні бінарні дерева (рис. 1.3). Особливістю та головною відмінністю повного бінарного дерева є те, що усі вершини, окрім листків, мають рівно по два дочірніх вузла. Усі листки також мають при цьому знаходитись на одному рівні (глибині).

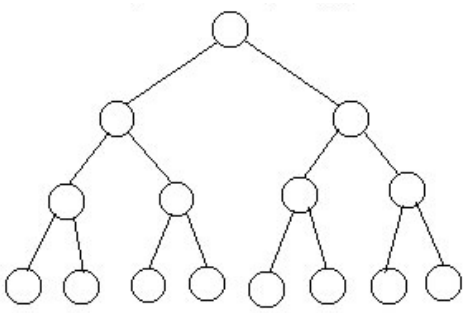


Рис 1.3. Повне бінарне дерево [1]

b) Неповні бінарні дерева (рис. 1.4). Головною відмінністю неповних бінарних дерев від повних є те, що у неповних вузли можуть мати нульову степінь на будь-якому рівні (глибині). [7]

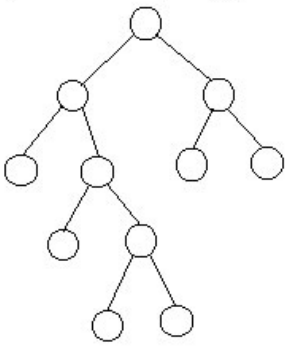


Рис 1.4. Неповне бінарне дерево

- N – арні дерева. Ця топологія, аналогічно до N – арного дерева, полягає у тому, що у кожного із батьківських вузлів відповідно може бути більше, аніж два дочірніх вузла. N – арні дерева, у свою чергу, можна поділити на дві категорії :

a) Неорієнтовані – це такі дерева, степені вершин у яких не перевищують N + 1.
b) Орієнтовані – це такі N – арні дерева, у яких число вихідних ребер (степені вершин) не перевищує N.

Важливо також зауважити, що дана топологія поєднує у собі властивості двох інших топологій, а саме шини та зірки.

Обмежену пропускну здатність топології можна цілком вирішити шляхом застосування так званого «товстого» дерева – Fat Tree (рис. 1.5).

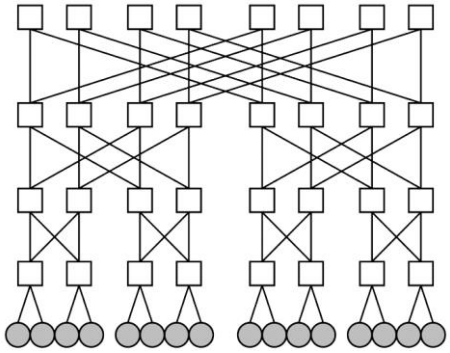


Рис 1.5. «Товсте» дерево (Fat tree)

Ця топологія є дуже ефективною для суперкомп'ютерів. Її суть полягає у тому, що відповідно зв'язки у такому дереві стають «товшими» та продуктивнішими з точки зору пропускної спроможності залежно від рівнів, між якими знаходиться зв'язок. Таким чином, чим ближче зв'язок знаходиться до кореня дерева, тим буде «товшим».

Маршрутизація у дереві

Топологія дерево може бути закодована наборами із нулів та одиниць. Як приклад можна розглянути укладку дерева на площині. Таким чином, починаючи із будь-якої вершини, ми будемо рухатися по ребрам нашого дерева, при цьому повертаючи у кожному вузлі у напрямку найближчого ребра справа та розвертаючись назад у кінцевих вузлах дерева. При цьому проходячи по деякому із ребер, будемо записувати 0 при русі по ребру у перший раз та 1 при русі по ньому у другий раз, тобто у зворотньому напрямку. Якщо при цьому m

– це число усіх ребер у дереві, то через кількість кроків 2m ми повернемося у той же самий вузол, при цьому пройдячи двічі по кожному із ребер.

1.1.4 Кільце

Кільце – це топологія, у якій сполучені вершини разом із ребрами зв'язку виглядають, ніби кільце (мають його форму). У найпростішій формі кільцевої топології кожна вершина сполучена лише із двома іншими. Найпростіша форма кільцевої топології зображена на рис. 1.6.

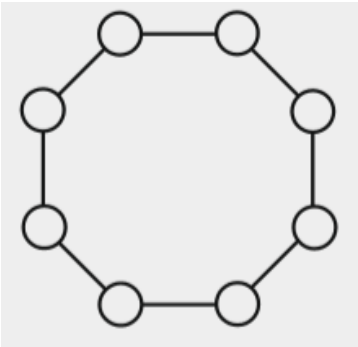


Рис 1.6. Найпростіша кільцева топологія [1]

Кільцеві топології можна поділити на два підвиди у залежності від кількості каналів, що сполучають два сусідніх вузла. Таким чином, виділяємо однонаправлені (один вузол між двома вузлами) та двонаправлені (два вузли між кожними двома вузлами).

Кільце є прикладом двовимірної статичної топології через дві причини. По-перше, між кожними двома вузлами у ній можливий лише один фіксований шлях, і по-друге, структура ієрархії зазвичай не змінюється динамічним шляхом.

Топологія кільце у найпростішому варіанті реалізації має наступні характеристики :

1. Діаметр : $[\frac{N}{2}]$
2. Кількість зв'язків : N
3. Зв'язність : 2
4. Розмір мережі : N;

5. Порядок вузла : 2
6. Ширина бісекції : 2 [1]

Кільцева топологія має як свої переваги, так і ряд певних недоліків. Почнемо з переваг :

- Простота топології
- Забезпечення стабільного і стійкого функціонування навіть при великому навантаженні на мережу

Недоліки ж у даної топології наступні :

- Вихід із ладу одного із вузлів цілком може позначитися на роботі усієї мережі
- Важко знайти місце виникнення проблеми при наявності деяких відхилень у роботі системи
- Найпростіший тип топології, у випадку розширення системи у майбутньому призводить до проблем через занадто великий діаметр.

Одним із можливих способів усунення останнього недоліку кільцевої топології – надто великого діаметра – є додавання додаткових зв'язків – хорд між певними вузлами топології. Утворена топологія називатиметься хордальною (рис. 1.7).

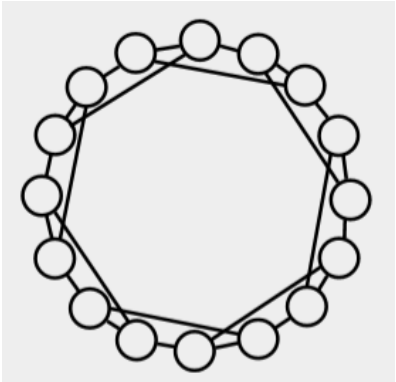


Рис 1.7. Хордальна топологія [1]

Хордальна топологія належить до тривимірних статичних топологій. Якщо з збільшувати порядок вузлів й надалі, ми зможемо досягти ще меншого

діаметра топології, та, як наслідок, пришвидшення передачі повідомлень у ній. Одним із варіантів, до якого можна прийти, є топологія із циклічним зсувом зв’язків (рис. 1.8).

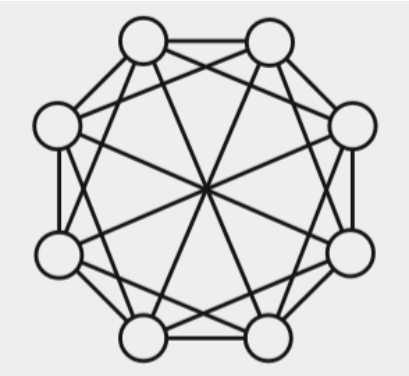


Рис 1.8. Топологія із циклічним зсувом зв’язків [1]

Топологія кільце може бути представлена у вигляді замкненого графа, який вміщує у собі один цикл. Даний граф C_k буде вміщувати у собі k вершин та k ребер. Цей граф може бути приведений до графа – зірки шляхом додавання вершини третього класа та видалення всіх існуючих ребер і додавання зв'язків між вершинами першого класа із доданою вершиною третього класа.

Кільце дуже часто використовується у побудові комп’ютерних мереж (рис. 1.9).

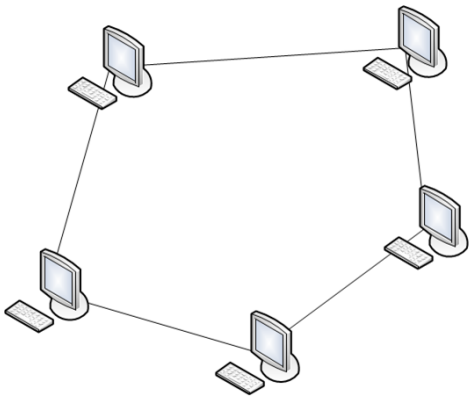


Рис 1.9. Комп’ютерна мережа у вигляді кільця [8]

У такій системі дані можуть передаватися в одному із доступних напрямків, або, як альтернатива, у двох напрямках відразу. Кожен із методів має свої плюси та мінуси, а який із них буде використовуватися зазвичай залежить від конкретної використовуваної технології у локальній мережі.

Маршрутизація у кільці

У кільці, на відміну від інших топологій, таких як зірка або шина, не використовується конкурентний спосіб передачі даних, і таким чином комп'ютер у мережі отримує дані від того, що стоїть попереду у списку адресатів та направляє їх далі, якщо він не має бути їх одержувачем. Перелік одержувачів генерується тим самим комп'ютером, що являється й генератором маркера. Мережевий модуль генерує сигнал, що зветься маркерним (зазвичай це близько 2-10 байт для недопущення затухання) та передає його до наступної системи (іноді за зростанням MAC – адреси). Наступна ж система, після прийняття сигналу не аналізує його, а банально передає далі. Це все зветься нульовим циклом.

1.1.5 Решітка

Решітка – це топологія, вузли якої утворюють зв'язки у вигляді одно- або багатовимірної решітки (рис. 1.10).

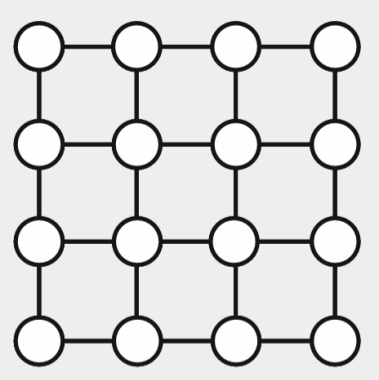


Рис 1.10. Приклад решітки [1]

При цьому кожне ребро такої топології-решітки має розташовуватися паралельно до її осі та поєднувати між собою два суміжних вузла, що відповідно знаходяться уздовж цієї осі. Решітка є ускладненим варіантом деяких інших топологій, наприклад, гіперкуба.Багатовимірна решітка, що до того ж є циклічно поєднана у декількох вимірах, називається топологією тор (рис. 1.11). Решітка є двовимірною статичною топологією. Для підтвердження цього можна навести два аргументи. Перший – як правило, її структура не

змінюється під час проведення операцій із нею. По-друге, між кожними двома вузлами у решітці є можливим лише один зафіксований шлях.

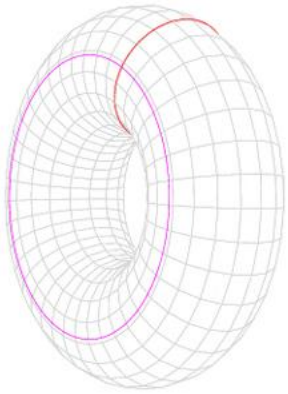


Рис 1.11. Топологія Тор

Топологія решітка із розмірністю $m * m$ $(m = \sqrt{N})$ матиме такі характеристики:

1. Діаметр : $2(m - 1)$
2. Кількість зв'язків : $2(\mathrm{N} - \mathrm{m})$
3. Зв'язність : 2
4. Розмір мережі : N;
5. Порядок вузла : 4
6. Ширина бісекції : $\sqrt{N}$ [1]

У випадку проведення над плоскою матрицею операції wraparound (згортання) у одному із напрямків, можемо отримати один із варіантів циліндричної топології. Перший із них - горизонтальний циліндр, зображений на рис. 1.12.

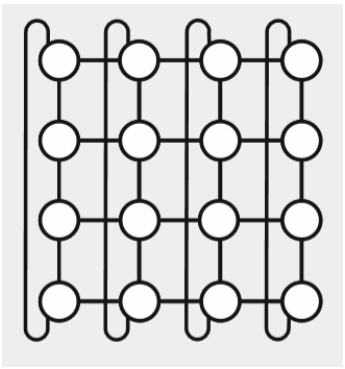


Рис 1.12. Горизонтальна циліндрична топологія

Другий варіант - вертикальний циліндр, зображений на рис. 1.13.

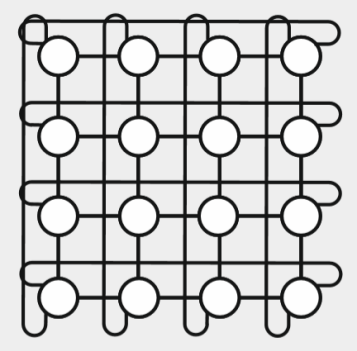


Рис 1.13. Вертикальна циліндрична топологія

У випадку проведення тієї ж операції wraparound (згортання) над плоскою матрицею, але вже по двох напрямках відразу, отримаємо топологію мережі, що зветься тороїдальною (рис. 1.14).

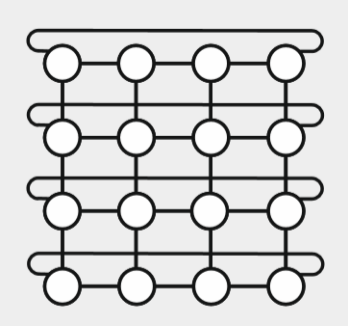


Рис 1.14. Тороїдальна топологія мережі

Маршрутизація у решітці

Мережі топології типу решітки при використанні більш ніж одного виміру будуть мати високий рівень надмірності маршрутів та зв’язків, проте потребуватимуть великої кількості зв’язків між вузлами топології. При цьому передача даних здійснюється із використанням транзитних вузлів, що збільшує рівень латентності та потребує адекватного вибору протокола для маршрутизації. Модифікації мережі, при якій дана топологія перетворюється у тор по одному чи декільком вимірам, мають менший діаметр, а, отже, також мають й більш низький рівень середньої латентності, проте й потребують декотрої кількості зв’язків більшої довжини, або ж згортання декотрих вимірів.

1.1.6 Повнозв’язна топологія

Повнозв’язна топологія – це топологія, усі вузли у якій напряму пов’язані один із одним. Дана топологія також часто згадується із назвами «максимальне угрупування» та «топологія кліка». Повнозв’язна топологія дозволяє перейти до мінімальних витрат для передачі інформації. Першим мінусом даної топології є те, що її може бути досить важко реалізувати за дуже великої кількості вершин (процесорів). Також можна зауважити, що дана топологія зазвичай не дає суттєвого покращення продуктивності системи, тому що кожна з операцій пересилання інформації потребуватиме аналізу з боку вузола-відправника всіх його N-1 входів. Тому, аби прискорити дану операцію, потрібно, аби всі входи принаймні могли аналізуватися паралельно, що, в свою чергу, суттєво ускладнює процес конструювання вузлів.

Повнозв’язна топологія відноситься до статичних тривимірних топологій за двома ознаками : по-перше, між двома вузлами у ній можливий лише один фіксований шлях, і по-друге, у процесі її використання структура ієрархії зазвичай не змінюється. [9]

Приклад повнозв’язної топології можна побачити на рис. 1.15.

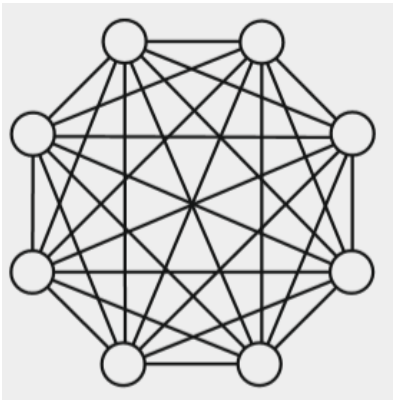


Рис 1.15. Повнозв’язна топологія [1]

Повнозв’язна топологія має такі характеристики :

1. Діаметр : 1
2. Кількість зв’язків : $\frac{N(N-1)}{2}$
3. Зв’язність : $N-1$

4. Розмір мережі : N;
5. Порядок вузла : $N - 1$
6. Ширина бісекції : $\frac{N^2}{4}$ [1]

Маршрутизація у повнозв'язній топології

Передача доступу у мережах побудованих за повнозв'язною топологією зазвичай відбувається за допомогою метода передачі спеціального маркера. Маркер – це певний пакет, у якому знаходиться спеціально побудована послідовність біт. Маркер послідовно передається у кожний вузол топології в одному напрямку. Таким чином, кожен вузол має ретранслювати маркер, що передається. Якщо вузол отримав пустий маркер, він може передати свої дані. Таким чином, цей маркер із даними передається доти, доки не знайдеться вузол – адресат. У цьому вузлі дані приймуться, а маркер буде рухатися далі та зрештою повернеться до відправника. Тільки після того, як вузол – відправник буде впевненим, що пакет доставлений до місця призначення, маркер буде звільнений. [10]

1.1.7 Лінійна топологія

Лінійна топологія – це проста топологія, вузли у якій формують одновимірний масив та розташовуються один за одним у ланцюжку. Лінійна топологія, таким чином, має складатися із двох крайніх вузлів та будь-якої кількості проміжних вузлів між ними. Приклад лінійної топології зображений на рис. 1.16.

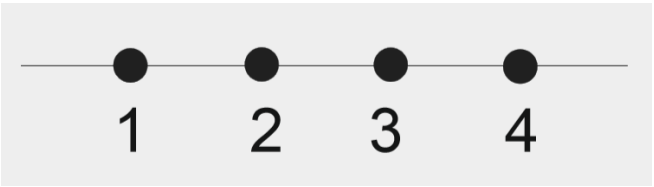


Рис 1.16. Лінійна топологія

Лінійна топологія відноситься до статичних одновимірних топологій, тому що по-перше, між двома вузлами у ній можливий лише один фіксований шлях, і по-друге, у процесі її використання структура її ієрархії не змінюється.

Лінійна топологія має такі характеристики :

1. Діаметр : $N - 1$
2. Кількість зв’язків : $N - 1$
3. Зв’язність : 1
4. Розмір мережі : N;
5. Порядок вузла : 2
6. Ширина бісекції : 1

Перевагами такої топології являються :

- Простота
- Низька вартість

Недоліками являються :

- Обмеження у функціональності мережі
- Обмеження у розмірі мережі
- Обмеження у розширюваності мережі

Маршрутизація у лінійній топології

В лінійної топології немає властивості повної симетричності, тому що вузли на кінцях топологї мають лише одну комунікаційну лінію. Тобто, їх степінь рівна одиниці, у той час як степінь інших вузлів є рівною двом. Час пересилання повідомлення залежатиме від відстані між вузлами, а відмова одного із цих вузлів може призвести до неможливості пересилання цього повідомлення. Із цієї причини у лінійних мережах використовуютсья відмовостійкі вузли, що можуть ізолювати себе від мережі при відмові, таким чином дозволяючи повідомленню оминати пошкоджений вузол.

1.2 Огляд евристичних алгоритмів

Далі будуть розглянуті такі евристичні алгоритми, що використовуються для пошуку маршрутів у топологіях, як генетичний, мурашиний, жадний та бджолиний алгоритми.

1.2.1 Мурашиний алгоритм

У колонії як правило немає домінуючих особин, тобто, немає ні керівників, ні підданих, немає також і лідерів, що могли б роздавати накази та координувати дії підлеглих. Колонія зазвичай є повністю самоорганізованою. При цьому кожна із мурах володіє тільки інформацією про локальну обстановку, і ні одна із них таким чином не має й представлення про ситуацію в цілому, а тільки про те, що дізналася безпосередньо сама або від своїх родичів, у явному вигляді або неявному.

Під час кожного проходу від мурашника до джерела їхньої їжі, та навпаки, мурахи залишають після себе доріжку із феромонів. Інші ж мурахи відчують такі сліди на землі та почнуть інстинктивно рухатися по ним. При цьому ці мурахи також лишатимуть після себе доріжки із феромонів, і тому логічним є зробити висновок, що чим більше різних мурах проходитиме по одному шляху, тим більш помітним та привабливим цей шлях ставатиме для їх родичів. Також є очевидним й те, що чим менший буде шлях від мурашника до джерела їжі, тим менше колонії мурах потрібно буде часу, а тому, тим й швидше сліди, що вони залишатимуть на шляху, ставатимуть помітнішими.

Як вже було сказано вище, мурашиний алгоритм працює за допомогою моделювання багатоагентної системи.

Кожна мураха при цьому зберігає у пам’яті перелік вузлів, які він вже пройшов. Тож коли мураха вибиратиме вузол для того щоб здійснити наступний крок, він буде пам’ятати про вже пройдені вузли та не буде розглядати їх як потенційно можливі вузли для переходу. Таким чином, із кожним кроком мурахи цей перелік поповнюється новим вузлом, а перед новою ітерацією алгоритму, тобто, перед наступним проходом мурахою шляху, він спустошуватиметься.

Також, окрім переліку заборон, для вибору вузла щоб перейти до нього, мураха керується й так званою «привабливістю» ребер, по яким він зможе пройти. Вона залишить від декотрих факторів. Перший із них – це вага ребра,

тобто, відстань між сполученими вузлами. Друге – це сліди феромонів, що були залишені на ребрі мурахами, які вже пройшли по ньому раніше. Тож очевидно, що, на відміну від ваг ребер, що завжди будуть константами, сліди цих феромонів оновлюватимуться із кожною ітерацією даного алгоритму : як і в природі, сліди випаровуються із плином часу, і при цьому мурахи, що проходять по ним, навпаки будуть їх посилювати. [11] Приклад класичного мурашиного алгоритму можна побачити на рис. 1.17.

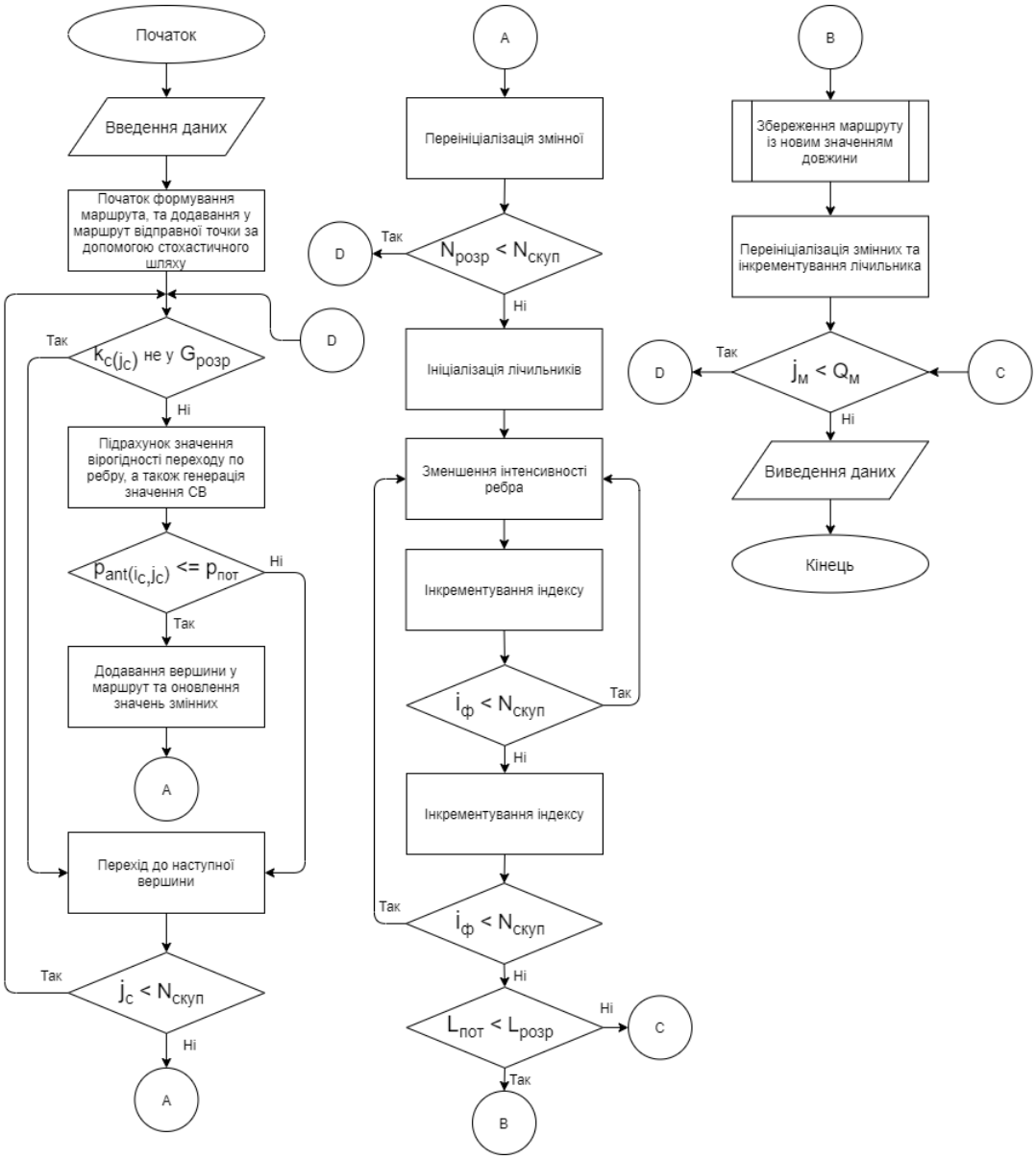


Рис 1.17. Блок-схема мурашиного алгоритму [3]

Як ми можемо побачити із діаграми мурашиного алгоритму, він не є надто простим та складається із великої кількості повторюваних інструкцій. Також

можна помітити, що при деяких параметрах мурашиний алгоритм може перетворитися у класичний жадний алгоритм, або може зійтися до субоптимального рішення. Таким чином, вибір правильного співвідношення вхідних параметрів є надвичайно важливим, і тому є об'єктом багатьох досліджень, а в загальному випадку є здійснюваним на основі власного досвіду.

1.2.1 Бджолиний алгоритм

Бджолиний алгоритм відноситься до природніх алгоритмів. До них можна також віднести й такі алгоритми як мурашиний та генетичний, алгоритм імітації віджигу, і ще декілька майже невідомих алгоритмів.

Тож надалі будуть пояснюватися принципи роботи бджолиного алгоритму. Він є алгоритмом синтеза та оптимізації, і насправді має дуже цікавий принцип роботи.

Для того, щоб зрозуміти принципи роботи бджолиного алгоритму, або, як його також називають, методу роя бджіл (МРБ), ми маємо вдатися до порівняння його з аналогією – реальним роєм бджіл.

Тож уявімо собі рій бджіл, що знаходиться у полі. На даний момент основна його ціль – це знайти на цьому полі таку область, де щільність квіток буде найбільшою. Бджоли, не маючи будь-якого представлення про це поле у будь-якому випадку, починають свій пошук квітів із випадкових точок та із випадковими векторами швидкості. Кожна із бджіл може при цьому пам'ятати певні позиції, де вона знайшла найбільшу кількість відповідних квітів із найбільшою щільністю. Вибираючи між поверненням до того місця, де бджола власноруч побачила найбільшу кількість відповідних квітів, та рухом до місця, що було знайдене іншими, як таке, де є найбільша кількість квітів, кожна бджола буде спрямовуватися у напрямку між цими двома точками, в залежності від того, що вплине більше на неї – соціальний рефлекс або персональні згадки.

Також під час мандрівки бджола може знайти місце з вищою щільністю квітів, ніж було знайдено до цього. Таким чином, у майбутньому це місце може

бути помічене як нове місце із найбільшою кількістю квітів, а також і як місце із найбільшою кількістю квітів, що було знайдене загалом всім роєм.

Бджола може й випадково пролетіти повз такого місця, де кількість квітів являється більшою, ніж було знайдено будь-якою бджолою із рою. Після цього весь рій спрямовуватиметься до цього місця, у додаток до власних відкриттів кожної окремої бджоли. [12] Приклад структури бджолиного алгоритму наведено на рис. 1.18.

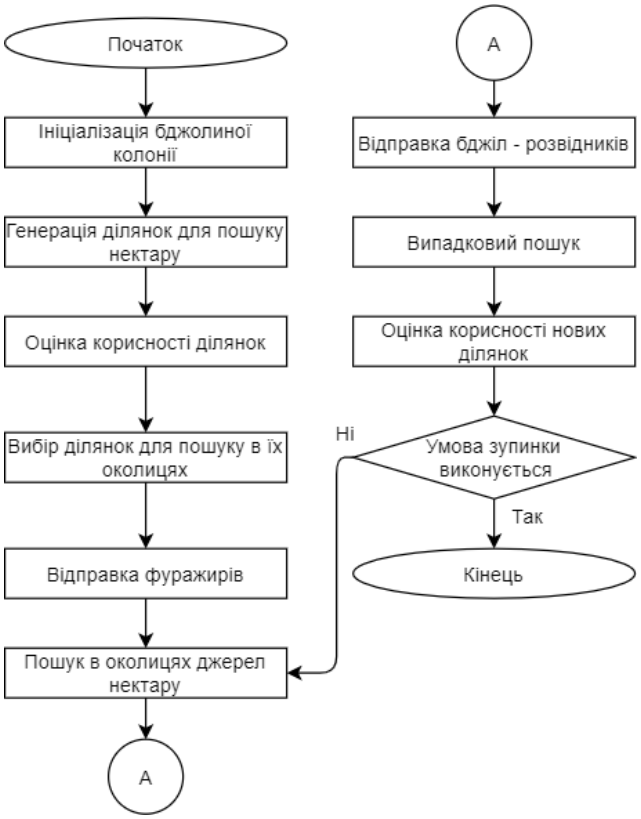


Рис 1.18. Структурна схема бджолиного алгоритму

Таким чином, бджоли досліджуватимуть поле, перелітаючи при цьому місця із найбільшою щільністю квітів, вони будуть сповільнюватись у їх напрямку. Також вони будуть безперервно перевіряти місця, що пролетіли, і порівнювати їх із вже знайденними раніше місцями із найбільшими щільностями квітів, при цьому сподіваючись знайти абсолютний максимум щільності квітів.

1.2.2 Генетичний алгоритм

Генетичний алгоритм використовує механізм еволюції, такий як природній відбір. Це означає, що більш перспективні особини у ньому матимуть набагато більше шансів на виживання та репродукцію. Завдяки передаванню генетичної інформації від батьків до дітей, ці діти наслідують головні властивості від своїх батьків. Тож потомки більш перспективних батьків також будуть більш перспективними, і процент їхньої присутності в популяції збільшуватиметься. Після послідовної зміни десятків або сотень поколінь, середній фітнес особин суттєво збільшується.

Генетичний алгоритм – це в першу чергу еволюційний алгоритм, або, іншими словами, його основна так звана фішка – це комбінування, що також зветься схрещуванням. Нескладно здогадатися, що ідея алгоритму, як і у попередніх прикладах, запозичена у природи. Тож шляхом перебору, та, найголовніше, природнього відбору, виходить так звана правильна комбінація. [13]

Генетичний алгоритм поділяється на три етапи :

- Схрещування
- Селекція
- Формування нового покоління

Якщо результат ітерації нас не влаштовує, ці етапи будуть повторюватися доти, доки результат нас не почне влаштовувати, або виповниться одна із термінальних умов :

- Кількість поколінь досягне встановленого максимуму
- Буде вичерпаним час на мутацію

Приклад узагальненого генетичного алгоритму можна побачити на рис. 1.19.

Вхідні дані для генетичного алгоритму :

- $G_{\text{скуп}}$ – множина всіх можливих маршутів між вершинами у графі;
- $N_{\text{рп}}$ – розмір множини $G_{\text{скуп}}$;

- $c_{\text{іт}}$ – індекс вершини у маршруті, може мати значення із діапазону : [1, $N_{\text{рп}}$];
- $i_{\text{г}}$ – індекс маршруту, може мати значення із діапазону : [1, $N_{\text{скуп}}$];
- c – змінна, що є лічильником; [2]

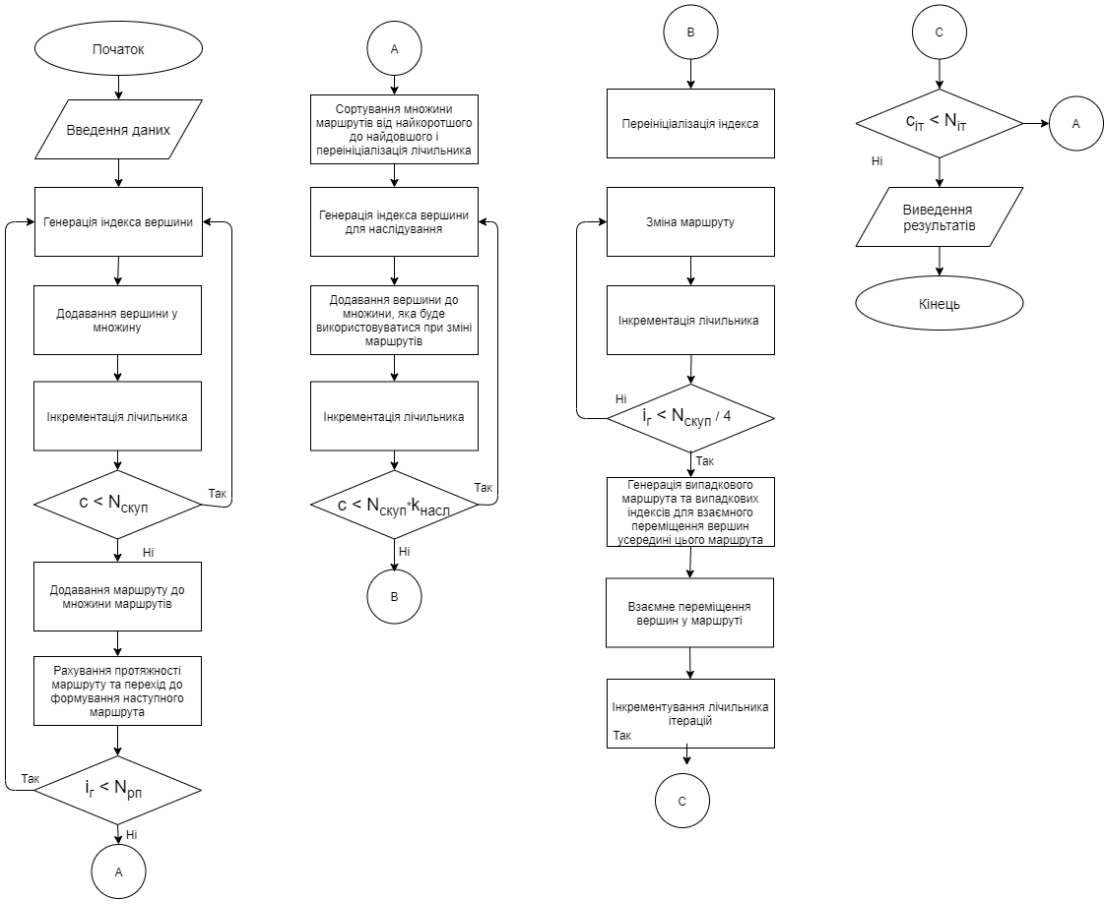


Рис 1.19. Блок-схема генетичного алгоритму [2]

Як видно із блок-схеми узагальненого генетичного алгоритму, у загальному він складається із великої кількості повторюваних кроків, і завершується зазвичай при виконанні однієї із двох умов – або вичерпання часу на мутацію, або досягання кількістю поколінь значення встановленого заздалегідь максимуму.

1.2.3 Жадібний алгоритм

Жадібний алгоритм полягає у знайденні локальних оптимальних рішень та побудові вирішення загальної проблеми із них. Приклад жадібного алгоритму зображений на рис. 1.20.

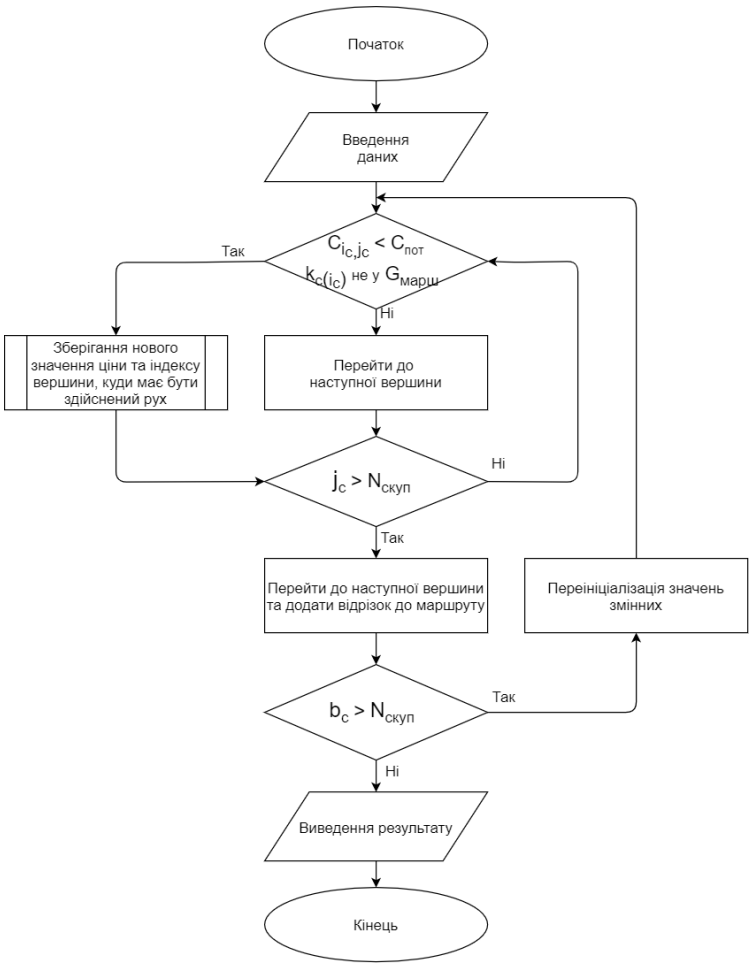


Рис 1.20. Блок-схема жадібного алгоритму

Вхідними даними для жадібного алгоритму являються :

- K_c – множина скупчень вершин у графі;
- b_c – лічильник вершин графа;
- i_c – індекс вершини – поточної початкової точки;
- j_c – індекс вершини – поточної кінцевої точки;
- $c_{\text{пот}}$ – значення найбільш вигідної ціни ребра у даний момент;
- $n_{\text{пот}}$ – індекс вершини із найвигіднішою ціною ребра;
- $G_{\text{м}}$ – вершини у результуючому маршруті; [4]

Головною перевагою цього алгоритму є, звичайно, час виконання, що є для нього дуже малим. Проте головним і не менш вагомим аргументом проти використання жадібного алгоритму є те, що рішення, отримане за його допомогою у багатьох випадках є дуже не оптимальним.

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі були розглянуті різні види основних топологій, такі як наприклад гіперкуб, зірка, дерево, кільце, решітка, повнозв'язна та лінійна топології. Окремо були розглянуті переваги цих топологій, були зображені та проаналізовані їх різні підвиди. Також були розглянуті найважливіші параметри кожної із топологій, такі як діаметр, кількість зв'язків, зв'язність, розмір мережі, порядок вузла та ширина бісекції. Ці параметри є дуже важливими при порівнянні різних топологій.

Також у даному розділі, а саме у підрозділах, пов'язаних із кожною топологією були розглянуті механізми маршрутизації, пов'язані із кожною із розглянутих топологій.

Виходячи із розглянутих топологій та їх аналізу, було вирішено для побудови системи у даній роботі використовувати повнозв'язний вид топології. Для цього є дві причини. Перша із них – те, що повнозв'язна топологія є досить поширеною у багатьох мережах, та знаходження шляхів у ній є досить актуальним завданням, яке виникає у великій кількості систем. Друга причина – те, що алгоритм, за яким відбуватиметься робота із знаходження шляхів у даній топології, буде цілком переносимим і на роботу з іншими типами топологій, із невеликими змінами у його основі.

Також у першому розділі були розглянуті основні евристичні алгоритми, які часто використовуються у розв'язанні багатьох задач, та у той же час які чудово зарекомендували себе як алгоритми, що підходять для знаходження маршрутів у різних топологіях.

Були розглянуті такі евристичні алгоритми, як бджолиний, мурашиний, жадібний та генетичний. Були розглянуті схеми їх роботи, а також принцип роботи кожного із цих алгоритмів. Були встановлені їх недоліки та переваги. Зокрема, було встановлено, що генетичний та мурашиний алгоритми мають найскладніші алгоритми роботи, але при цьому й працюють із великими обсягами вхідних даних, видаючи при цьому найкращі результати. Жадібний

же алгоритм, наприклад, працює швидше за інших, проте видає не настільки стабільно хороші результати.

У об’єкті розробки даної роботи – системи пошуку маршрутів було прийнято рішення використовувати два найкращі по двом параметрам алгоритми. Перший із них – це генетичний алгоритм, він був вибраний через те, що показує себе найкраще у роботі із великими об’ємами вхідних даних, якщо ми говоримо про точність результатів. Другий же вибраний алгоритм – це жадібний, а саме алгоритм найближчого сусіда, що є не настільки ефективним з точки зору точності результату, проте є дуже хорошим у результатах за параметром витраченого часу, адже навіть із вхідними даними великого об’єму даний алгоритм справляється дуже швидко.

РОЗДІЛ 2
ОГЛЯД ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ СИСТЕМИ

2.1 Мова програмування

Для використання у даній роботі була обрана мова програмування Java, цьому сприяло багато факторів, що будуть послідовно надані надалі.

2.1.1 Мова Java

Java є однією із мов загального призначення. При цьому вона активно слідує об’єктно-орієнтовній парадигмі програмування. Загалом, своєрідним дивізом цієї мови програмування можна назвати «Напиши один раз, а потім використовуй усюди». Це призвело до того, що Java на сьогоднішній день де тільки не використовують : мережеві, десктопні, корпоративні, мобільні додатки – все це пишеться із використанням даної мови.

Java – це не просто мова програмування, а також й ціла екосистема із інструментів. Ці інструменти фактично охоплюють майже усе, що тільки може знадобитися, щоб програмувати на мові Java. Тож до неї входять :

- JDK (Java Development Kit) – це так званий комплект розробника на Java. Загалом, за допогомою JDK та стандартного блокноту вже є можливим написання та компіляція й запуск програми на Java.
- JRE (Java Runtime Environment) – це виконуюча система Java. Загалом кажучи, це мінімальна реалізація віртуальної Java-машини, необхідна для запуску програм, написаних на цій мові програмування. Сама ж дана виконуюча система складається із автономної віртуальної машини Java (JVM), певних інструментів налаштування та стандартної бібліотеки класів, що зветься Java Class Library.
- IDE (Integrated Development Environment) – це є інтегроване середовище для розробки. Іншими словами, це такі інструменти, що допомагають нам запускати, редагувати та компілювати наш код.

Найпопулярнішими середовищами розробки у екосистемі мови Java є Intellij IDEA, Eclipse, а також NetBeans.

Код, написаний на Java, буквально можна знайти усюди. Вона є основною мовою для розробки додатків на Android. Також вона активно використовується у веб-застосунках, певних веб-сайтів навіть державного масштабу, та ба більше – навіть у технологіях обробки big data, таких як наприклад Apache Storm, або Hadoop. Також можна сказати, що Java чудово підходить й до наукових проектів. Якийсь час мова Java домінувала й на ринку мобільних додатків, ще до появи смартфонів, адже перші мобільні ігрі написані у нульових роках, були розроблені саме на ній.

Завдяки своїй дуже довгій історії, безперечно, Java заробила собі місце у залі слави програмування. Проте це не означає, що на сьогоднішній день її популярність почала спадати. Наприклад, на рис. 2.1. приведений індекс TIOBE, у якому показана розрахована популярність програм у світі використовуючи результати пошукової видачі.

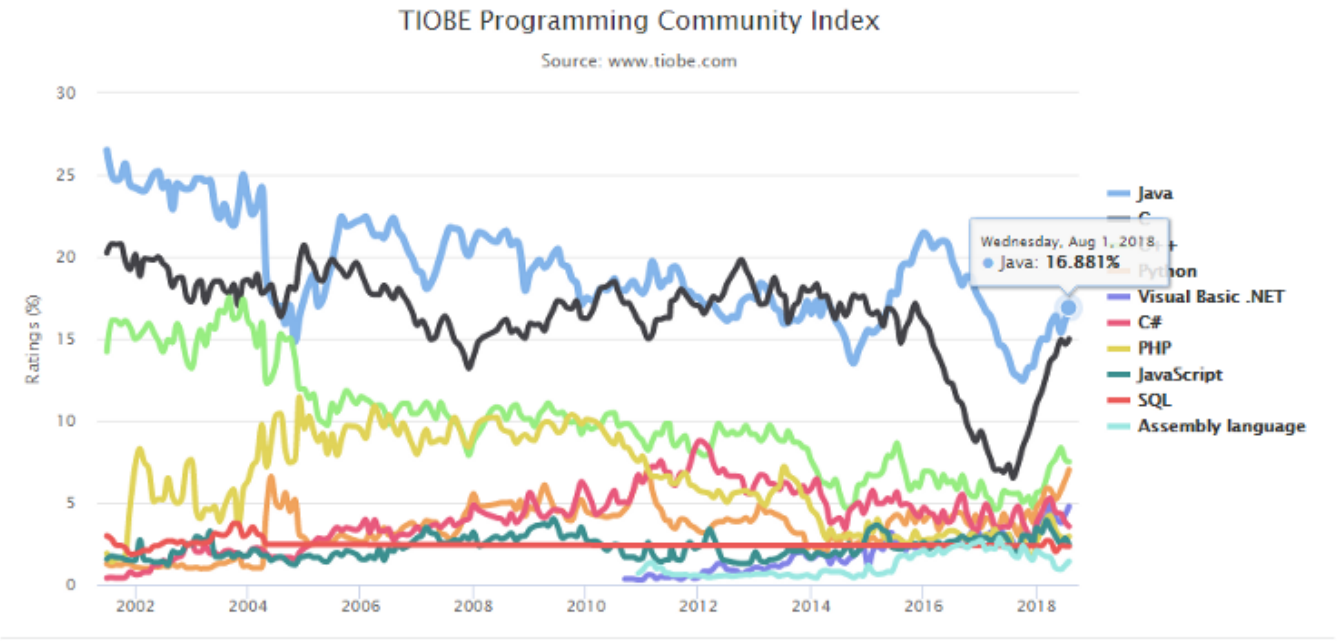


Рис 2.1. Індекс TIOBE, серпень 2018 р.

Як видно із графіка, популярність мов Python та Go на даний момент зростає, але Java все ж стабільно залишається на вершині цього переліку вже більше, ніж десятиріччя.

Історія мови Java розпочалася у 90-х роках минулого сторіччя із того, що у команді Sun Microsystems вирішили розробити покращену версію C++, що не буде залежати від платформи, а також буде доволі зручною для починаючих програмістів, і до того ж матиме автоматичне управління пам'яттю. Сама ж назва нової мови програмування – лише одна із запропонованих, яких було багато десятків. На сьогоднішній день логотип компанії є непримітним, але у той же час і дуже впізнаваним – це чашка із кавою. І вже не можна навіть точно стверджувати, що було першим – асоціація із Java або одержимість більшості програмістів кавою.

Першою назвою мови Java була Oak, що зберігалася до 1995 року, потім вона стала називатися Green, а вже тільки після цього набула своєї нинішньої назви. Історія ж створення цієї мови бере свій початок у далекому 1991 році, коли дехто під ім'ям Джеймс Гослінг створив свій проект для використання в одному із численних інших проектів.

Проте офіційною ж датою створення даної мови вважається 23 травня 1995 року, цього дня компанія Sun випустила перший реліз мови під версією 1.0. Також вона задекларувала девіз «Пиши один раз, запускай де завгодно», при цьому й забезпечуючи невеликою ціною на досить популярних платформах.

Через майже 11 років, у 2006-му, та ж компанія Sun вже випустила велику частину із коду мови у відкрите користування, як вільне програмне забезпечення у відповідності GPL, тобто до General Public License.

Восьмого ж травня 2007 року почався новий виток історії мови Java. Цього дня компанія завершила усі процеси, необхідні для того, щоб код мови став безоплатним та відкритим для усіх, окрім тієї невеликої частини з коду, авторських прав на котрий у компанії не було.

2.1.2 Переваги мови Java

- Java є об’єктно-орієнтованою мовою програмування. У ній усе, що завгодно, є об’єктом. Також завдяки цьому програми є легко розширювані.
- Java не є залежною від платформи. На відміну від багатьох інших мов програмування, таких як наприклад C або C++, Java із самого створення компілюється не у платформі зав’язаній на якусь конкретну машину, а у незалежний від платформи байт код. Останній може розповсюджуватись через інтернет, а також інтерпретується у JVM(тобто, Java Virtual Machine), у якій він у даний момент часу запущений. Процес виконання вихідного коду у Java показаний на рис. 2.2.

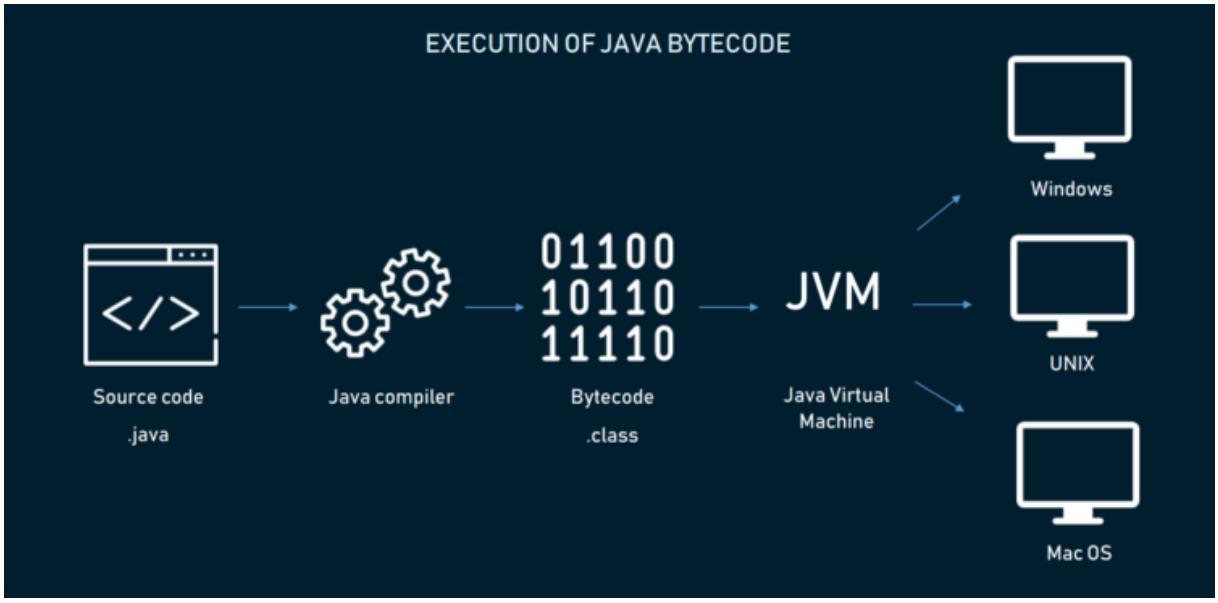


Рис 2.2. Процес виконання коду на Java

- Java є доволі безпечною. У ній перевірка достовірності виконується за допомогою шифрування на основі відкритого ключа.
- Java є архітектурно нейтральною. Це означає, що компілятор у ній генерує нейтральні архітектурно об’єкти формату файла, це робить можливим виконати скомпільований код на великій кількості різних процесорів, які мають у собі систему Java Runtime.

- Java є портативною мовою програмування. Такою її робить незалежність від реалізації різних аспектів специфікацій та архітектурна нейтральність. При цьому й компілятор у Java є написаним на C, ANSI із можливістю чистої переносимості, що також являється підмножиною POSIX-у.
- Java є доволі «міцною». Сама мова програмування докладає багато зусиль для того, щоб допомогти користувачу позбавитися від помилок у різних ситуаціях, в основному роблячи це у процесі компіляції, а також і перевірку помилок під час виконання програми.
- Java є доволі простою. Це означає, що процеси вивчення, а також базового введення у курс мови програмування Java є доволі простими, і для того, щоб освоїти дану мову, від вас у першу чергу потребується лише знання основних елементарних концепцій об'єктно-орієнтовного програмування, можливо навіть на одній із інших мов. Для прикладу, написання базової програми показане на рис. 2.3.

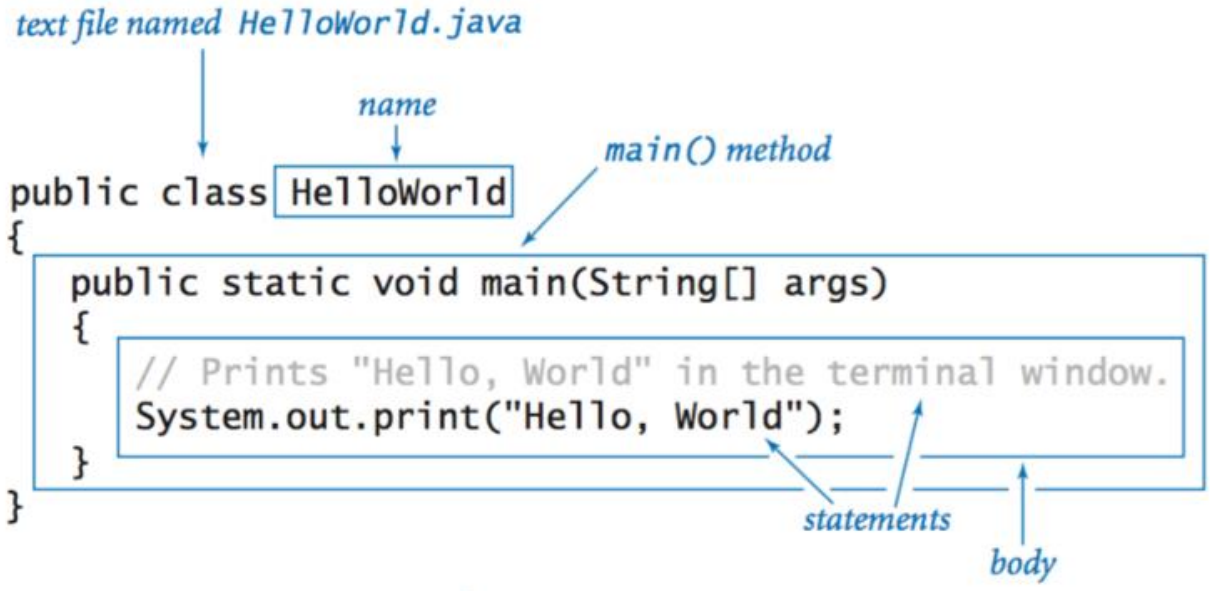


Рис 2.3. Структура базової програми на Java

- Java є інтерпретованою мовою. Це означає, що при її виконанні байт код, отриманий із вихідного коду програми, «на льоту» транслюється в машинні інструкції та ніде не зберігається при цьому. Це допомагає зробити даний процес аналітичним та швидким, оскільки так званий біндінг відбувається додатково, і із невеликою вагою процесу.
- Java є високопродуктивною. Для цього був введений так званий Just-in-time компілятор, що й дозволило отримати зазначену високу продуктивність компіляції.
- Java є дуже розповсюдженою. Ця перевага надто сильно відчувається, коли виникає потреба пофіксити якусь помилку у коді. Якщо вона виникла у якійсь менш популярній мові програмування, часто можна зіштовхнутися із проблемою, коли знайти правильне рішення може зайняти багато часу, або, ще гірше – не принести результату. Із Java ж такого майже не буває. Через те, що мова є дуже популярною, відповідь на 95 відсотків питань з приводу різних помилок при її застосуванні можна легко або не докладаючи надто великих зусиль, знайти в інтернеті.
- Java є багатопотоковою мовою програмування. Завдяки цьому із її використанням можна писати програми, що можуть бути, по-перше, виконуваними одночасно декільком потоками, що може у багатьох ситуаціях пришвидшити її виконання та / або надати переваги у обчислювальних можливостях програми, а по-друге, дозволяє розбити програму на незалежні частини, що будуть виконуватися різними потоками, що знову-ж таки призведе до збільшення можливостей написаної програми. Приклад багатопотокового виконання програми зображений на рис. 2.4.

- Java є доволі динамічною. Програмування на ній вважається більш динамічним, ніж, наприклад, на тих же C або C++, так як сама мова з самого початку розроблялася із призначенням відповідати умовам, що постійно змінюються. [15]

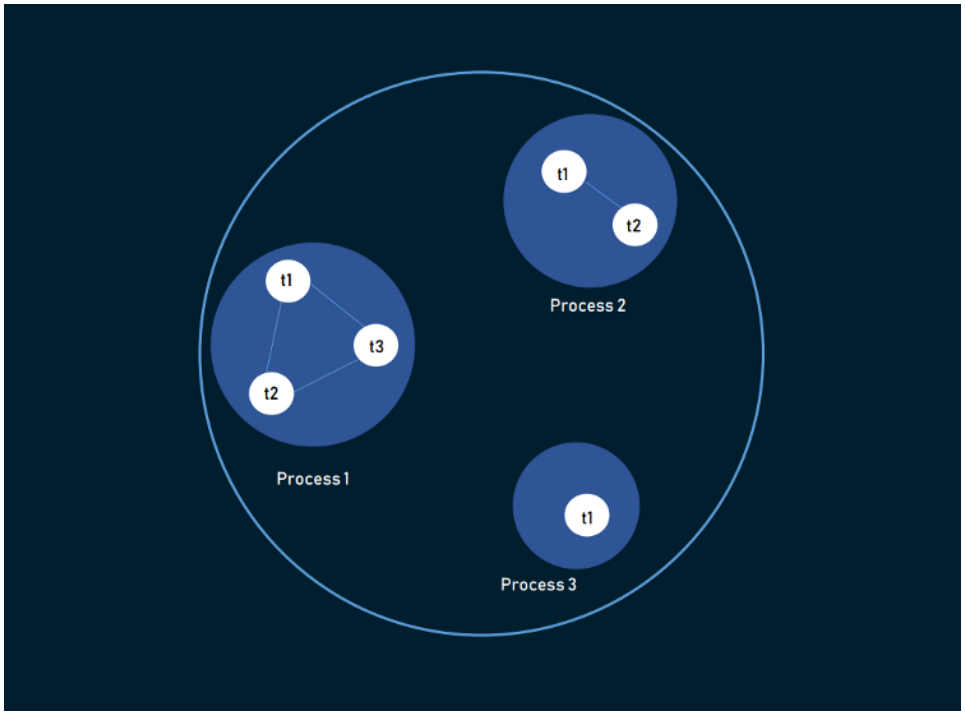


Рис 2.4. Приклад виконання багатопотокової програми

2.1.3 Недоліки мови Java

- Низька продуктивність. Як і у будь-якої мови високого уровня, цілком логічно, що у даної мови програмування продуктивність в цілому є не високою. Це відбувається через компіляцію та абстракцію за допомогою віртуальної машини. Проте й це не єдині причини низької швидкості мови Java. Іншою є, наприклад, очистка пам’яті. Ця функція є надзвичайно корисною, проте у той же час призводить до великих проблем із продуктивністю, у тому випадку, якщо вона потребує не менше ніж 20 відсотків часу процесора. Також занадто сильне використання пам’яті може бути викликане поганим налаштуванням кешування. Наступною проблемою є можливе взаємне блокування потоків. Ця ситуація відбувається,

коли декілька з потоків намагаються отримати доступ до одного і того ж ресурса програми. У цьому випадку може відбутися помилка через нестачу пам’яті. Проте, усі ці проблеми є вирішуваними шляхом правильної побудови програми.

- Відсутність нативного дизайну. Для створення графічних інтерфейсів GUI розробники вимушені використовувати різні інструменти, що залежать від конретної мови. Наприклад, для додатків під Android використовують Android Studio, що допомагає створювати додатки із відповідним нативним дизайном. Проте, коли наприклад справа доходить до юзер інтерфейсу на ПК, такого інструменту у Java екосистемі немає.
- Важкий та багатослівний код. Багатослівність коду на перший погляд може показатися перевагою мови програмування, що може допомогти при її вивченні. Проте на практиці черезмірно довгі конструкції тільки ускладнюють читання та перегляд коду. Як і природні мови, так і мови програмування високого рівня можуть вміщувати у себе зайву інформацію. Гарно ілюструє даний недолік рис. 2.5.

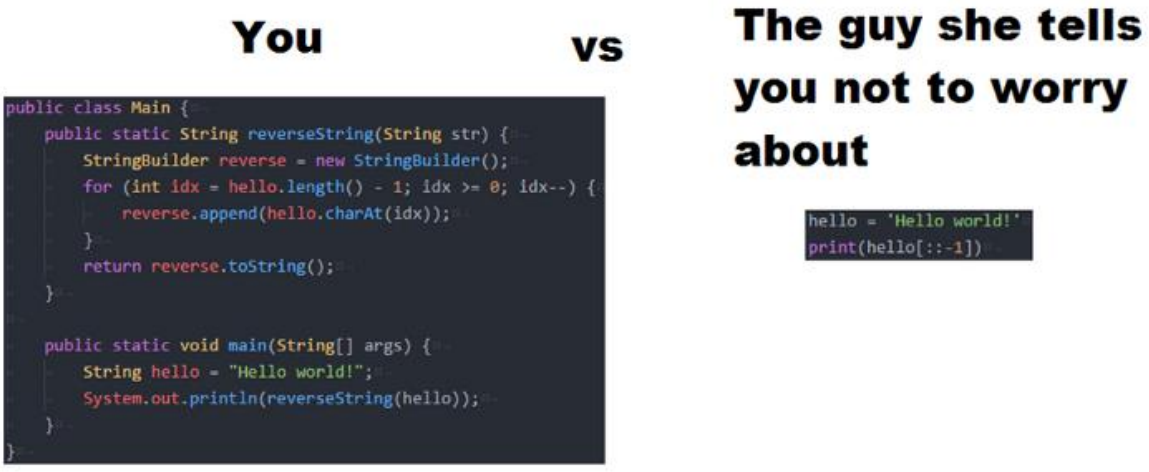


Рис 2.5. Порівняння одного і того ж фрагмента написаного на Java та Python

При порівнянні на даному рисунку фрагменту коду на цих двох різних мовах, бачимо очевидну перевагу у даній ситуації з боку лаконічного коду на пайтоні. [14]

2.1.4 Підсумок щодо вибору мови Java

Підсумовуючи усі зважені «за» та «проти» аргументи щодо використання мови Java у нашому проекті можна прийти до наступного висновку, що вона цілком відповідає вимогам, що ставляться перед майбутньою програмою та цілком може використовуватися для її написання таким чином. Варто додати, що кількість переваг, що надаються при її використанні, значно перевищує кількість її недоліків.

2.2 Фреймворки та бібліотеки

Далі описуються фреймворки та бібліотеки, які будуть використовуватися у проекті.

2.2.1 Spring

Spring - це дуже популярний фреймворк, що служить для полегшення та покращення якості розробки додатків на Java. Головний плюс даного фреймворку – те, що він дозволяє полегшити розробку J2EE застосунків для розробників. Перейдемо до переваг, що надає Spring при своєму використанні. Ось основні з них :

- Spring власноруч будує каркас нашого майбутнього застосунку, так би мовити «заготівку» для нашого майбутнього додатку. [16] При цьому цуй фреймворк й диктує нам певні умови та правила побудови цього застосунку, адже є певна архітектура застосунку, під яку нам необхідно буде підлаштувати свою бажану функціональність. Ця функціональність в цілому й буде бізнес-логікою у нашому додатку. До складу Spring входить багато дочірніх проектів, що є специфічними для деякої функціональності. Наприклад, Spring MVC, Spring Data, Spring Security та інші. Із цих

проектів розробник може вибрати той, що необхідний йому у конкретній ситуації та для конкретної задачі, а інші при цьому не використовувати, це і є головним принципом модульної побудови застосунку. Структуру фреймворку Spring можемо побачити на рис. 2.6.

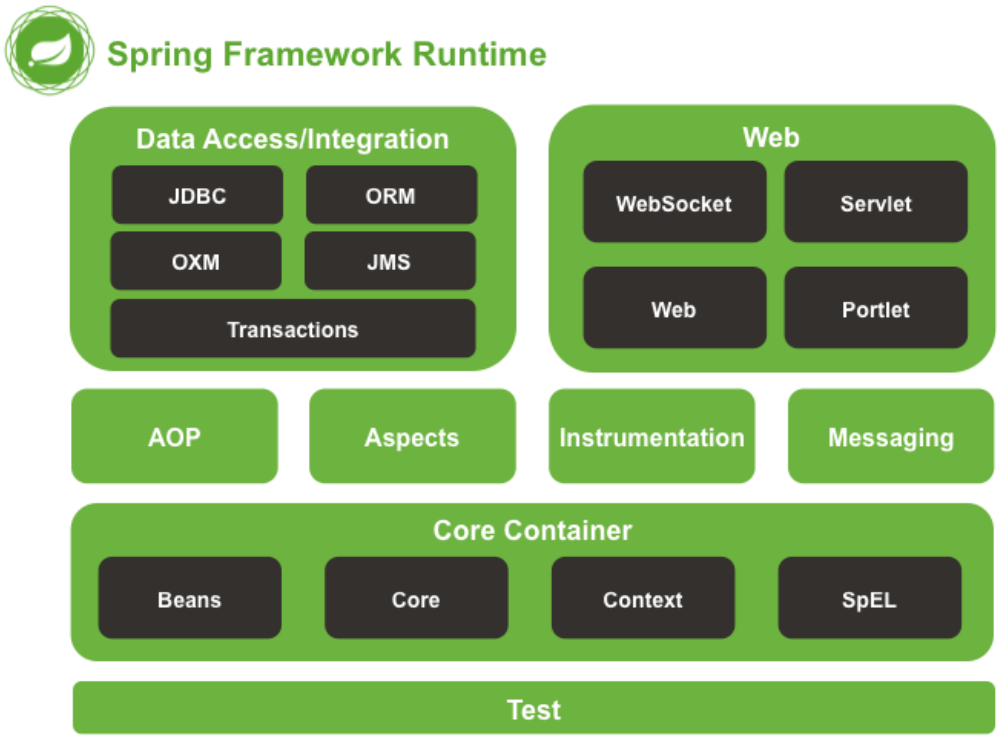


Рис 2.6. Структура Spring фреймворку [23]

- У додатку, розробленому за допомогою Spring, усі класи та об’єкти будуть слабко зв’язаними за рахунок використання такого механізму, як впровадження залежностей. Власне, це й було однією із головних цілей створення Spring – розірвати залежності одних об’єктів від інших. Що ж таке залежність? Це таке відношення, коли перший об’єкт використовує методи другого об’єкту, тобто, перший об’єкт залежить від другого об’єкту, чиї методи він використовує. А чому ж він залежить від нього? Тому, що доки другий об’єкт не буде створений, перший об’єкт не зможе реалізовувати свою функціональність. То як же розірвати таку залежність? Необхідно у перший об’єкт «впровадити» посилання на

другий об’єкт, наприклад через конструктор, або сеттер. Цей процес і є таким, що називається впровадженням залежності. При цьому дуже важливо пам’ятати, що в фреймворку Spring об’єкти необхідно будувати відштовхуючись від інтерфейсів, щоб залежності могли впроваджуватися у вигляді інтерфейсу для можливості подальшої заміни реалізації.

- Нам не треба буде створювати власноруч об’єкти із використанням оператора new, адже цю функцію ми делегуємо контейнеру Spring. Це називається IoC (інверсія контролю), що означає передачу функції інстанціювання необхідних залежностей, тобто об’єктів, контейнеру. Яка ж тоді роль розробника у цьому? Власне, оголосити компонент, для того, щоб він попав до так званого контексту Spring. Цей контекст насправді просто є мапою, де знаходяться усі біни. Тому коли хтось каже, що бін знаходиться у контексті даного фреймворка, можна вважати, що він лежить у мапі, а Spring просто знає ключ для того, щоб його звідти дістати. При цьому усі класи, що відмічені як біни в xml – конфігурації або у класах за допомогою анотації @Component, інстанціюється та кладеться у мапу, яка виглядає як Map<key, bean> map1, тобто, у контейнера власне є така мапа, у яку він складає усі біни (це ключове поняття у Spring – бін, що є об’єктом, що менеждиться спрінгом. Тож для того, щоб звичайний клас став менеджед спрінгом, ми маємо вказати спрінгу, що необхідно його додати у контекст). При необхідності впровадження даного біну, контейнер робить щось навподобі map1.get(key), і при цьому у якості ключа виступає тип поля.
- Spring звільняє нас не тільки від необхідності створювати об’єкти, але і від їх пов’язування. Нприклад, анотація @Autowired дозволяє

нам автоматично зв’язувати наші компоненти. Спрингову анотацію @Autowired можна було б і описати дуже просто – ніби ми просимо контейнер спринга подивитися у його мапу із бінами та знайти у ній такий бін, який імплементує потрібний нам інтерфейс або являється інстансом потрібного класу. Якщо такий бін знайдеться, то необхідно покласти його у те поле, перед котрим написана дана анотація. Таким чином, автоматичне зв’язування компонентів дозволяє зменшити кількість нашого коду при декларуванні залежностей компонентів.

- У Spring налаштування наших компонентів є відділеними від програмного коду. Винесення конфігурації, тобто, управління залежностями, в окремий файл суттєво полегшує усілякі подальші зміни у проекті, як наприклад заміна реалізацій :
    1. Покращення можливості тестування. Коли класи проектуються на основі інверсії залежностей та інтерфейсів, проста заміна залежностей їхніми моками стає цілком можливою при тестуванні.
    2. Можливість програмувати у декларативному стилі за допомогою використання анотацій суттєво зменшує кількість коду в додатку.
    3. Підтримка та дуже гарна інтеграція із різними технологіями доступа до даних, транзакцій і т.д, а також аспектно-орієнтоване програмування, що покращує та полегшує розробку.
    4. Чудова документація, що дуже сильно допомагає при відладці коду.

Тож, як бачимо, даний Spring Framework вражає навіть невеликою кількістю своїх основних переваг (все це без урахування його специфічних частин). Тому,

пишучи програму за допомогою екосистеми Java, ми не пройдемо повз нього та використаємо його у розробці нашої програми.

2.2.2 Maven

Maven – це інструмент, що служить для автоматичної збірки проектів. В основному він використовується саме Java-розробниками, проте є й окремі плагіни для інтеграції із такими мовами, як Ruby, C, C++, Scala, а також й PHP та інші різні мови.

Зібрати простий проект навподобі «Hello, World» можна й використовуючи командний рядок, проте чим складніше ставатиме наше розроблена програма і чим більше вона використовуватиме всіляких сторонніх бібліотек або ресурсів, тим суттєво складніше й ставатиме команда для збірки. З іншої сторони, є Maven, який був розроблений якраз для таких задач.

Однією із найголовніших переваг даного фреймворку є можливість декларативного описання проекту. Для розробника в першу чергу це означає те, що йому не потрібно буде приділяти свою увагу усім аспектам у збірці, тому що всі необхідні для цього параметри вже будуть налаштовані за замовчуванням. Розробнику ж потрібно вносити певні зміни лише тоді, коли він хоче певним чином змінити стандартні налаштування. [17]

Базову структуру основного файлу Maven'а – pom.xml – можна побачити на рис. 2.7.

Наступна перевага даного проекту – це можливість дуже гнучкого управління залежностями. Maven має можливість завантажувати до локального репозиторію різні бібліотеки, здайснювати обробку транзитивних залежностей, вибирати необхідну версію пакета і т.і. Можна сказати також, що розробники фреймворку вирішили зробити його незалежним від операційної системи. Від платформи залежатимуть тільки параметри командної стрічки, але при цьому Maven дозволяє не звертати увагу й на цю деталь.

```xml
<project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

<groupId>com.mycompany.app</groupId>
<artifactId>my-app</artifactId>
<version>1.0-SNAPSHOT</version>

<name>my-app</name>
<!-- FIXME change it to the project's website -->
<url>http://www.example.com</url>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
</properties>

<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <pluginManagement>
        <plugins>
            <...>
        </plugins>
    </pluginManagement>
</build>
</project>
```

Рис 2.7. Базова структура pom.xml [17]

Якщо потрібно, система збірки може бути настроєна під бажання розробника, використавши вже готові архетипи та плагіни. Якщо ж підходячого не знайшлось, можна написати свій.

Можна також додати, що Maven є інтегрованим у всі популярні середовища розробки.

2.2.3 Spring Boot

Із плином часу разробники Spring вирішили надати розробникам-користувачам їх фреймворку такі утиліти, які автоматизуватимуть процедури налаштування системи та водночас можуть прискорити процеси створення та розгортання спрінг-базованих застосунків. Названі ці утиліти були Spring Boot.

Тому Spring Boot – це дуже корисний проект, який допомогає суттєво спростити створення та розробку додатків із використанням Spring. Ця частина фреймворку дозволяє нам створити веб-застосунок найбільш простим шляхом,

при цьому й потребуючи від розробників мінімуму зусиль спрямованих на налаштування та на написання коду.

Spring Boot має дуже великий функціонал, але найбільшими його перевагами є управління залежностями, вбудовані контейнери сервлетів, а також автоматична конфігурація.

Для того, щоб прискорити процес щодо управління залежностями, Spring Boot неявно запаковує ті залежності, які необхідні для кожного типу додатку на основі Spring та надає можливість розробнику користуватися ними у вигляді starter пакетів, таких як наприклад, spring-boot-starter-web, spring-boot-starter-data-jpa та інших. [18]

Ці так звані starter пакети являються набором зручних дескрипторів залежностей, що можно включити у наш додаток. Це допомагає нам отримати зручне та універсальне рішення для усіх Spring-based технологій, при цьому це й звільняє програміста від необхідності у зайвому пошуку прикладів потрібного коду та самостійного вилучення із них необхідних дескрипторів залежностей. Приклад таких залежностей показаний на рис. 2.8.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Рис 2.8. Імпортовані залежності Spring Boot [18]

Якщо ми, скажімо, хочемо використовувати Spring data jpa у нашому проекті, то ми можемо просто додати необхідну залежність spring-boot-starter-data-jpa та не шукати велику кількість залежностей, що сховані за цією.

Тож, іншими словами, Spring Boot бере на себе збір усіх загальних залежностей та дескрайбить їх у одному місці, що у свою чергу дозволяє розробникам просто брати та користатися ними, замість того, щоб кожного разу винаходити велосипед.

Із цього виходить те, що використовуючи Spring Boot наш файл із залежностями буде вміщувати суттєво менше стрічок, аніж якщо не користуватися цією можливістю Spring-екосистеми.

Spring Boot після включення вибраного стартер-пакету до списку залежностей намагатиметься автоматично налаштувати наш застосунок, на основі тих jar-залежностей, що були додані нами.

Наприклад, при додаванні до проекту spring-boot-starter-web, Spring Boot сконфігурує для нас автоматично такі зареєстровані біни : ResourceHandlers, MessageSource, DispatcherServlet та інші.

Також будь-який додаток, що будується на основі Spring Boot, включає у себе вбудований веб-сервер. Список контейнерів, що підтримуються «із коробки», налічує усі найпопулярніші із них.

Розробники теперь не мають піклуватися про налаштування контейнера сервлетів, а також про розгортання застосунку на ньому. З цього часу додаток може запускатися сам, так само як виконуваний jar-файл, і все це з використанням вбудованого сервера. [24]

Якщо ж нам треба буде використовувати окремий HTTP-сервер, то ми можемо просто для цього виключити дефолтні залежності. Також Spring Boot надає нам окремі стартери для різних HTTP серверів.

2.2.4 JUnit

Junit – це такий фреймворк, який був розроблений для тестування програм, що написані із використанням мови Java. Також можна сказати, що саме він лежить в основі розробки за допомогою тестування, тобто Test-Driven

Development (TDD), та до того ж ще входить до сімейства тестувальних фреймворків під назвою xUnit.

Приклад створення тестів за допомогою JUnit можна побачити на рис. 2.9.

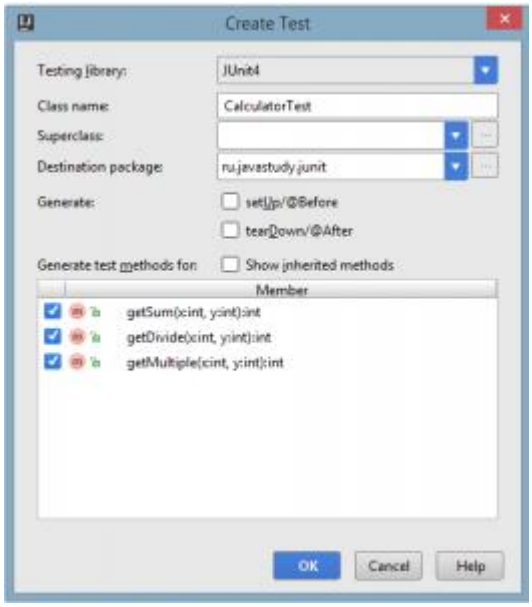


Рис 2.9. Приклад створення тестів за допомогою Junit [25]

З самого початку головною ідеєю даного фреймворку була фраза «Спочатку тести, а вже потім код». Цей підхід під собою має на увазі те, що спочатку ми маємо описати те, що хочемо отримати у результаті написання коду, а потім вже після написаних тестів на кожен кейс, що перевірятимуть правильність результатів виконання коду, пишемо сам необхідний код для того, щоб ці тести проходили. Такий підхід дозволяє значно збільшити ефективність роботи розробників та дозволяє писати більш красивий та структурний код, відразу думати про апі методів. Загалом, ми отримуємо й набагато меншу кількість часу, що витрачається командою на відладку програми. [19]

Також можна відмітити такі основні властивості даного фреймворку :

- Відкритий вихідний код, що може використовуватися як для написання коду, так і виконання тестів
- Є дуже простим у використанні
- Підтримує анотації для ідентифікації методів

- Дозволяє нам писати код більш швидко та більш якісно
- Дозволяє нам у майбутньому не вагаючись вносити зміни у програму, адже ми зможемо перевірити за допомогою юніт тестів, чи вона ще працює
- Можливість огранізовувати тести у зв’язки, що звуться тестовими сюїтами
- Має візуальну індикацію тестів, що можуть бути як червоними (ті, що не пройшли), так і зеленими (відповідно, ті, що пройшли)

Тож таким чином, JUnit являється невід’ємною частиною програм, написаних на мові Java, і ми не можемо дозволити собі не використовувати його у даній програмі, адже його переваги очевидні.

2.2.5 Swagger

Swagger, по суті, це фреймворк для специфікації RESTful API. На рис. 2.10. ми можемо побачити приклад використання Swagger UI.

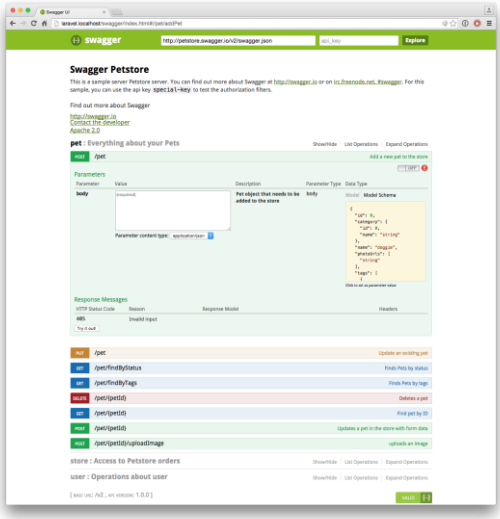


Рис 2.10. Приклад використання Swagger UI [20]

Основна його перевага заключається у тому, що він нам дає можливість не тільки інтерактивно переглядати специфікацію, що, до речі, вже є дуже великою цінністю, а також і дозволяє відправляти запити на наш апі.

Як бачимо, із використанням цього фреймворку ми досягли повного описання наших методів, що включає у себе також й моделі, коди відповідей, а

ще й параметри запитів. Таким чином, вся інформації про наш апі зібрана в одному місці та виглядає дуже інформативно. Через всі ці причини ми використовуватимемо Swagger UI у нашому проекті. [20]

2.2.6 Lombok

Lombok – це чудовий помічник розробника, який реалізує новій функції мови, не потребуючи при цьому значних зусиль із боку розробника. Звісно, кожному буде легшим встановити плагін, ніж навчити усіх інженерів у команді новій мові та портувати вже написаний код. Lombok не може абсолютно все, але вже навіть із коробки він дуже добрий помічник. Проявляється його дія у тому, що ми навішуємо анотації замість великої кількості коду, а сам ломбок за нас вже генерує необхідний код.

Приклад коду із використанням Lombok можемо побачити на рис. 2.11.

```
@Value
@Builder(toBuilder = true)
public class User {
  @NonNull
  UUID userId;
  @NonNull
  String email;
  @Singular
  Set<String> favoriteFoods;
  @NonNull
  @Builder.Default
  String avatar = “default.png”;
}
```

Рис 2.11. Код із використанням Lombok [21]

Ще одна з переваг даного фреймворку заключається у тому, що він зберігає відповідність кодових баз. Це в свою чергу значно полегшить масштабування команд та знизить навантаження у випадку переключення контексту, щоб запустити новий проект. Lombok, до речі, працює із будь-якою версією джави, починаючи із шостої. Тому можна розраховувати на його можливе використання майже у будь-якому проекті.

Врешті-решт, весь код, що генерується даною утилітою, можна було б писати власноруч. Проте Lombok нам допомагає спростити написання нудних частин кодової бази, при цьому не впливаючи на бізнес-логіку. Це нам дозволить сфокусуватися на речах, що є дійсно важливими для нас та нашого бізнесу, а також на найбільш цікавих задачах для програмістів. Можна додати, що одноманітний шаблонний код – це пуста трата часу більшості розробників. Тож якщо він не пишеться вручну, то вірогідність допустити помилку у ньому також падає. [21]

Тож у нашому проекті ми використовуватимемо даний фреймворк, оскільки ми не хочемо витрачати час на написання шаблонного коду.

2.2.7 MapStruct

Простими словами, MapStruct – це зібрані засоби для відображення бінів у мові Java. Цей API вміщує у себе такі функції, що автоматично відображатимуться між двома компонентами у Java. З допомогою MapStruct, нам необхідно лише створити інтерфейс, а сама бібліотека завдяки цьому вже створюватиме конкретну реалізацію потрібного нам мапера на етапі компіляції програми. Приклад простого мапера із використанням MapStruct можна побачити на рис. 2.12.

```
@Mapper
public interface SimpleSourceDestinationMapper {
    SimpleDestination sourceToDestination(SimpleSource source);
    SimpleSource destinationToSource(SimpleDestination destination);
}
```

Рис 2.12. Приклад мапера із використанням MapStruct [22]

Для більшості додатків ми можемо помітити велику кількість бойлерплейт коду, що перетворює одні POJO в інші. Наприклад, як правило, загальний тип перетворення відбувається між об’єктами із постійною підтримкою та так званими DTO (Data transfer object), що повертаються на та приймаються зі сторони клієнта. [22]

Тому основна проблема, вирішувана за допомогою MapStruct, це скорочення коду та передавання обов’язків щодо написання реалізацій потрібних нам маперів на цей фреймворк. Бібліотека робить це автоматично.

Щоб здійснювати перетворення одних типів даних у типи даних інших рівнів, ми будемо використовувати цю утиліту, оскільки вона допоможе нам як покращити структуру коду, так і скоротити час на його написання.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі були розглянуті технології, що будуть використані для написання системи – об’єкта розробки. Також були наведені обґрунтування, навіщо ці технології використовуватимуться та які переваги вони нададуть при їх використанні.

Перш за все, було розглянуте питання із вибору мови програмування для написання нашого проекту. Спочатку було декілька варіантів, проте найбільш серйозним та вірогідним із них стала мова Java, яка була детально розглянута у даному розділі. Зокрема, тут розглядається загальна інформація про дану мову програмування, багато її переваг та недоліків. Виходячи із цього, формується підсумок, із якого видно, що дана мова чудово відповідає вимогам нашого проекту та приймається рішення із її використання, адже у цьому випадку розроблювана система набуде багато переваг, що передадуться на неї відповідно до мови Java.

Також у другій частині даного розділу були розглянуті фреймфорки та бібліотеки, що використовуватимуться у даній роботі для написання вихідного коду. Усі вони виконуватимуть більшу чи меншу ролі, але усі ці ролі є важливими для даного проекту.

Першим із розглянутих фреймворків став Spring, який лежить в основі написання нашого проекту. Були розглянуті переваги, що надає використання цього фреймворку та було прийнято рішення про його обов’язкове використання у нашій системі, адже воно значно спростить написання, читабельність та підтримку нашого майбутнього додатку.

Наступним був розглянутий під-фреймворк Spring під назвою Boot. Були розглянуті дуже великі переваги цього фреймворку і було вирішено його використовувати у проекті для тих же цілей, що й його батьківський фреймворк.

Далі був розглянутий такий фреймворк як JUnit, який у великій мірі допоможе тестуванню нашого додатку і який на даний момент є майже незамінним у галузі тестування Java-додатків.

Наступним був розглянутий збірник Maven, який здатен значно прискорити процес розробки додатку, а також полегшити читабельність та переносимість об’єкту розробки на інші пристрої.

Далі ми розглянули Swagger, що дуже допоможе майбутнім клієнтам нашого API орієнтуватися у тому, як потрібно його використовувати та які ендпоїнти є у нашому проекті.

Передостанньою була розглянута бібліотека MapStruct, що є однією із основних маппер-бібліотек у світі Java. Ця бібліотека здатна допомогти у мапінгу наших об’єктів.

Останньою була розглянута бібліотека під назвою Lombok. Було показано, що вона здатна значно спростити написання коду проекту, та по більшій мірі прибрати взагалі написання «бойлерплейт» коду.

РОЗДІЛ 3
ДЕТАЛІ РОЗРОБКИ СИСТЕМИ

3.1 Розробка загальної архітектури системи

Система, що розроблюється у даній роботі, будується за архітектурою мікросервісного типу. Також архітектура розробляємої системи наслідує принципи архітектури REST.

Загальна структура системи зображена на рис. 3.1.

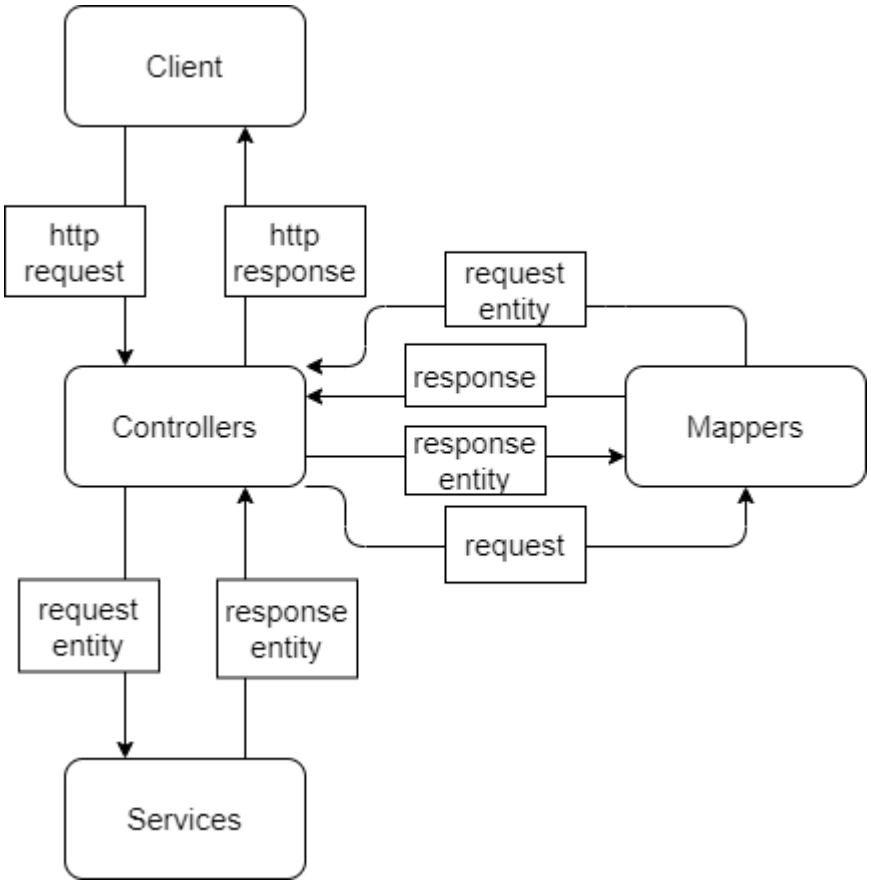


Рис 3.1. Загальна структура системи

Як бачимо із даної схеми, система складається із трьох основних блоків.

Перший із них – це блок REST-контролерів (Controllers), який представляє із себе API нашої системи та через який безпосередньо проходить уся взаємодія із клієнтами додатку. У ньому виконуються такі операції як отримання запитів за певними endpoint-ами, неявне їх перетворення у об’єкти, потім – перевірка отриманих об’єктів на відповідання встановленим правилам та стандартам API, логування запиту (для того, щоб системний адмін або хтось

інший міг потім в логах знайти помилку, або просто впевнитися, що все працює, як потрібно), виклик маперів для перетворення об’єктів у внутрішні сутності та виклик методів сервісів, після цього об’єкт, що прийшов як результат із сервісу, відправляється до мапера, а результат цієї операції буде загорнено у об’єкт ResponseEntity та відіслано у вигляді HTTP-відповіді із JSON-ом усередині клієнтові. Для більшої наглядності на схемі зображений й умовний клієнт даного API. Як можна побачити із схеми, клієнт проводить взаємодію із REST-контролерами, а саме із певним набором endpoint-ів наших контролерів за допомогою посилання HTTP запитів та отримання від них відповідей у вигляді комбінацій HTTP-заголовків та HTTP-тіл, із яких формуються HTTP-відповіді. Зокрема, для взаємодії із API у даній системі був обраний формат JSON, оскільки він є одним із найпопулярніших та найзручніших серед усіх аналогів.

Другим блоком у системі являється блок маперів (Mappers). Цей блок служить, по-перше, для маппінгу об’єктів запиту, що приходять як HTTP-запити у вигляді JSON від клієнтів, у внутрішні сутності програми. По-друге, мапер перетворяє й сутності, що повертаються із сервісів, в об’єкти відповідей. Це необхідно головним чином для того, аби підвищити рівень безпеки нашого додатку. Це виходить із того, що ми максимально не хочемо показувати кожному клієнтові структуру нашої системи. Замість цього ми видаємо за допомогою Swagger йому список endpoint-ів та описання того, що вони роблять, а він вже маючи тільки цю інформацію починає взаємодіяти із нашою системою, адже все, що йому потрібно власне це що він може отримати та що він має для цього надіслати на endpoint. Тому мапінг є невід’ємною частиною інкапсуляції та безпеки нашої системи.

Останнім і найголовнішим блоком у нашій системі є блок бізнес – логіки (Services). У ньому зосереджена вся основна логіка роботи нашої системи. Це власне пошук шляху у топології, а також вирахування статистики із обчислення, таких параметрів як час, який зайняло обчислення, а також власне й довжину вирахуваного шляху. При цьому варто зазначити, що сервіси власне

працюють тільки із об’єктами сутностей, тобто, вони не є зав’язаними на якісь фреймворки та власне є переносими із проекту в проект. Тому вони й приймають у аргументи сутності, й віддають інші сутності до контролерів.

3.2 Розробка директорно-пакетної структури системи

Оскільки у проекті використовується Maven – система збірки, то було прийнято логічне рішення використовувати наступну структуру вихідних директорій та пакетів :

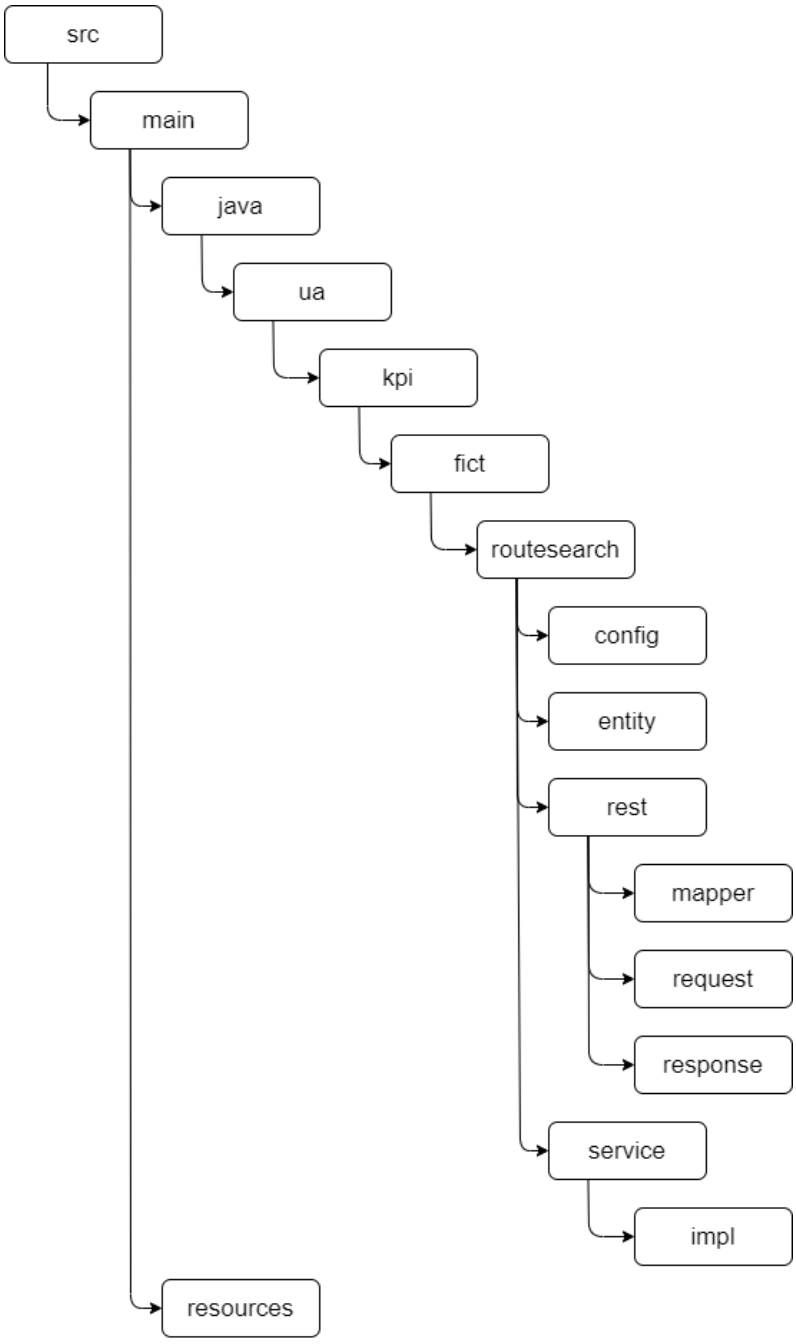


Рис 3.2. Директорно – пакетна структура системи

Як бачимо на схемі, на якій до речі зображена тільки основна частина системи (Тобто, на ній відсутня гілка src/test, у якій знаходяться тести), структура системи є доволі типовою для мікросервісного Maven-проекту. На вершині усієї вихідної ієрархії директорій знаходиться директорія src. Вона у собі ж містить дві інші директорії – main та test, другу із яких ми не будемо розглядати у цьому розділі. Директорія main містить також дві директорії. Перша із них – це директорія java, у якій повинні знаходитися вихідні файли із java-кодом. Друга – це директорія resources, яка є також стандартним елементом структури проектів Maven. У ній зазвичай знаходяться файли ресурсів, що не є безпосередньо основним кодом додатку. До них належать конфігураційні файли, файли із властивостями (properties), файли налаштування різних утиліт та фреймворків і т.і.

У директорії java знаходиться лінійна ієрархія пакетів ua.kpi.fict.routesearch, у останньому із яких знаходяться усі java-файли проекту. Пакет routesearch має такі підпакети :

- Config – пакет, у якому знаходяться налаштування проекту (у цьому проекті там тільки налаштування, пов’язані із документацією)
- Entity – пакет, який вміщує в собі класи внутрішніх сутностей програми
- Rest – пакет, який вміщує в собі рівень взаємодії із клієнтами додатку. Тут знаходяться такі підпакети як mapper, що міщує в себе класи маперів, request, у якому знаходяться моделі запитів із вказанням певних критеріїв для вважання їх об’єктів правильними, а також пакет response – тут знаходяться моделі об’єктів – відповідей, які користувачі API отримуватимуть як результат свого запиту.
- Service – пакет, де знаходиться бізнес-логіка проекту. На першому рівні лежать інтерфейси та абстрактні класи, тобто, загалом контракти, реалізації яких можуть варіюватися за потреби, а також тут

знаходиться підпакет impl, у якому лежать синглтонові реалізації даних інтерфейсів.

Таким чином, усі розглянуті вище пакети мають свої назви та місця у директорно-пакетній ієрархії проекту не просто так, а кожен із них має свою роль, що має бути зрозумілою тому, хто відкриє цей проект уперше.

3.3 Розробка модуля бізнес-сутностей

Модуль бізнес-сутностей системи знаходиться у проектному пакеті ua.kpi.fict.routesearch.entity. Він вміщує у собі класи сутностей, які використовуються безпосередньо у здійсненні бізнес-логіки програми. У модулі знаходяться такі класи :

- Point – це основний фундаментальний клас сутності, що відображає точку у двовимірній координатній системі. Відповідно, у об’єктів цього типу мають бути два поля :
  - Private int x – координата по осі x
  - Private int y – координата по осі y
- Route – це один із основних класів сутностей, який відображає маршрут побудований із точок у двовимірній координатній системі. Даний клас застосовується для більш зручної та читабельної взаємодії із маршрутом, аніж просто із списком із точок. Усередині нього знаходяться три поля :
  - Private List<Point> points – список точок у двовимірній системі координат, що входять у маршрут. Для зберігання множини цих точок вибраний саме List, адже нам важливо, щоб зберігався порядок їх знаходження у маршруті. Також можна помітити, що використана змінна саме типу List, що дозволить нам згодом змінити імплементацію списку, не змінюючи клієнтський код.
  - Private Double fitness – це числове значення, яке відображає фітнес цієї особини (у даному випадку маршруту).

- Private Integer distance – це числове значення, яке відображає загальну дистанцію цього маршруту.

- Population – це один із основних класів сутностей бізнес-логіки системи. Він відображає популяцію особин із реального світу, яка застосовується у генетичному алгоритмі. Фактично, популяція у даному контексті це список маршрутів, тому усередині класу лежить всього одне поле :
  - Private List<Route> routes – це список маршрутів, що належать даній популяції. Ця популяція приймає участь у роботі генетичного алгоритму.
- RouteSearchInputData – це клас основної сутності вхідних даних для вирахування маршруту. Він відображає множину точок, що передаються у систему для того, щоб побудувати із них найкоротший маршрут. Тому у даній сутності всього одне поле :
  - Private List<Point> points – це множина точок, що передаються на вхід до сервісів із метою знаходження шляху, що пролягає між ними. Для їх зберігання обраний саме тип списку, оскільки він достатньо швидкий, а також тому що надалі потрібно буде отримувати елементи із нього по індексу, а в цьому найкращий саме список ArrayList.
- GenericRouteSearchInputData – це клас сутності вхідних даних для вираховування маршруту, специфічних саме для генетичного алгоритму. Ці дані можуть знадобитися для конфігурування параметрів генетичного алгоритму. Даний клас наслідує клас RouteSearchInputData задля уникнення дублювання коду та використання поліморфізму у сервісах. Також у ньому додаються нові поля для об’єктів :
  - Private Double mutationRate – це число, яке використовуватиметься як шанс мутації, потрібний у генетичному алгоритмі для того, щоб задати шанс випадкової мутації утворених чайлдів.

- Private Integer tournamentSize – це число, яке використовуватиметься як розмір турніру у генетичному алгоритмі, тобто, для того, щоб визначити, скільки перентів будуть приймати участь у турнірі для вибору найперспективніших із них.
- Private Integer populationSize – це число, яке використовуватиметься як розмір популяції у генетичному алгоритмі, тобто як кількість маршрутів, що міститимуться у кожній популяції у генетичній еволюції.
- Private Integer populationsInEvolutionCount – це число, яке використовуватиметься як кількість популяцій для еволюції у генетичному алгоритмі, тобто як показник, коли алгоритму потрібно зупинитися для того щоб знайти найкращу поточну дорогу та віддати її як результат.
- Private Boolean elitismEnabled – це число, яке використовуватиметься як показник використання чи не використання елітних особин у генетичному алгоритмі, тобто, чи потрібно при утворенні нової популяції якусь кількість особин із найвищим фітнесом переносити відразу у нову популяцію.
- Private Integer elitePointsCount – це число, яке використовуватиметься як кількість елітних точок у генетичному алгоритмі, тобто, кількість точок із найвищим фітнесом, які відразу переміщуватимуться у нову популяцію, ще до турнірів, схрещування та мутації. Дана кількість не матиме значення, якщо використання елітних особин вимкнено.

- RouteSearchResult – це клас сутності результату обчислення найкращого маршруту за допомогою сервісів. Таким чином, він повертається як результат відповідних сервісів обчислення маршруту. У даному класі знаходяться два поля :

- Private Route route – це поле, яке відображає маршрут, який був знайдений у сервісі. Маршрут у собі вміщує список точок, значення дистанції маршруту, а також його фітнес, що використовувався у обчисленнях.
- Private Long millisecondsTaken – це поле, яке відображає кількість мілісекунд, яку зайняло обчислення даного результату вирахування маршруту.

Таким чином, бачимо, що усі класи-сутності, що знаходяться у даному пакеті, безпосередньо пов’язані із бізнес-логікою системи. Вони не будуть відомі користувачам цієї системи, що добре для безпеки системи. Також вони майже не зав’язані на фреймворки. Все що в них використовується, окрім Java, це Lombok, який просто допомагає зменшити кількість бойлерплейт коду та не є фреймворком таким чином.

3.4 Розробка модуля бізнес-логіки

Модуль бізнес-логіки системи знаходиться у проектному пакеті ua.kpi.fict.routesearch.service. Він вміщує у собі інтерфейси – контракти основних бізнес-сервісів, а також у пакеті impl вміщує реалізації цих інтерфейсів.

На першому рівні модуля знаходяться такі інтерфейси :

- PointService – це інтерфейс, який призначений для опису поведінки сервісу, що працює із операціями що безпосередньо відносяться до типу Point. Усередині є всього один метод із сигнатурою calculateDistance(Point, Point), що повертає значення типу double :
  - Double calculateDistance(Point firstPoint, Point secondPoint) – метод сервісу точок, який служить для прорахування відстані між двома переданими параметрами-точками у двовимірній системі координат. Використовується у даній системі іншими

сервісами, винесений у даний сервіс для дотримання принципів SOLID, а саме принципу єдиної відповідальності.

- RouteService – це інтерфейс, який призначений для опису поведінки та контракту сервісу, що працює із операціями, які безпосередньо відносяться тільки до сутності Route. Усередині є чотири абстрактних методи, і усі вони виконують певні дії, приймаючи на вхід об’єкти типу Route :
    - Route findFittest(List<Route> routes) – метод сервісу маршрутів, який приймає на вхід список із маршрутів. У самому методі відбувається знаходження маршруту із найбільшим значенням фітнесу у цьому списку. Цей маршрут повертається як результат методу.
    - List<Route> findSeveralFittest(List<Route> routes, int quantity) – метод сервісу маршрутів, який знаходить задану кількість quantity маршрутів із найбільшим фітнесом у списку routes. Даний метод приймає два параметри, власне список маршрутів та кількість маршрутів для вибірки. Повертає метод список вибраних маршрутів.
    - Double getFitness(Route route) – метод сервісу маршрутів, який повертає фітнес маршруту, що був переданий до методу за допомогою аргументу.
    - Int calculateDistance(Route route) – метод сервісу маршрутів, який повертає загальну дистанцію маршруту, що був переданий до методу як аргумент.
- RouteSearchService – це інтерфейс, який призначений для опису поведінки сервісу, що вираховує маршрут у топології. Тому даний інтерфейс має всього один абстрактний метод, що власне й має

вираховувати маршрут, але при цьому цей метод у майбутньому може бути, наприклад, перевантажений :

- RouteSearchResult findShortestRoute(RouteSearchInputData inputData) – метод сервісу пошуку маршруту, який служить для того, щоб знаходити найкоротший маршрут для вхідних даний (на даний момент це список точок) та повертати об’єкт типу RouteSearchResult, який має у собі вирахуваний маршрут, дистанцію, фітнес та час, за який була здійснена дана операція.

Також на першому рівні пакета є й один абстрактний клас :

- TimingRouteSearchService – це абстрактний клас, реалізовуючий шаблон «Шаблонний метод». Він реалізовує інтерфейс RouteSearchService та оверрайдить його єдиний абстрактний метод, у якому викликається його власний абстрактний метод. Таким чином у оверрайдженому методі знаходиться логіка, спільна для усіх нащадків даного абстрактного сервісу. Виходячи з назви даного класу, ця спільна логіка є логікою ведення обліку часу, за який виконується операція пошуку маршруту.
  - @Override Public RouteSearchResult findShortestRoute (RouteSearchInputData inputData) – метод абстрактного сервісу, що оверрайдить метод інтерфейсу RouteSearchService, тобто у даному випадку додає до сигнатури метода тіло. У даному методі здійснюється виклик другого метода класу та за допомогою Clock ведеться облік часу, за який викликаний метод відпрацює.
  - Protected abstract Route findRoute(RouteSearcInputData inputData) – це абстрактний метод даного класу, який

безпосередньо має містити логіку знаходження маршрута у наслідників даного абстрактного класу-сервіса.

Усередині модуля сервісів знаходиться й пакет із імплементаціями impl. Усередині нього знаходяться чотири імплементації сервісів :

- GeneticRouteSearchService – спадкоємець класу TimingRouteSearchService, що реалізує його абстрактний метод із знаходження маршруту, і також має низку приватних методів :
  - updateGeneticPropertiesIfNeeded() – якщо є потреба, оновляє значення генетичних констант.
  - setGeneticPropertiesFromInputData() – оновлює генетичні константи згідно до введених даних.
  - generateRandomRoutes() – генерує випадкові маршрути із списку точок і кладе їх у популяцію.
  - performEvolution() – проводить еволюцію над вказаною кількістю популяцій.
  - evolve() – проводить еволюцію над поточною популяцією.
  - provideInheritanceInPopulation() – проводить наслідування у популяції за допомогою схрещування двох батьків, вибраних за допомогою турнірів.
  - performMutationInPopulation() – проводить мутацію у популяції.
  - crossover() – проводить схрещування між двома точками.
  - initializeRoute() – ініціалізує маршрута – спадкоємця.
  - addPointsFromFirstParent() – додає до спадкоємця випадкові точки із першого з батьків.
  - addPointsFromSecondParent() – додає до спадкоємця усі точки, що залишилися, від другого з батьків.
  - mutate() – мутує вибраний маршрут.

- selectByTournament() – проводить турнір між визначеною кількістю маршрутів, та вибирає переможця із найбільшим фітнесом серед них.

- GreedyRouteSearchService – спадкоємець класу TimingRouteSearchService, який реалізує його абстрактний метод із знаходження маршруту, а також має низку приватних методів :
  - chooseFirstPointAndMoveItToResult() – вибирає випадковим чином першу точку та кладе її у маршрут-результат.
  - getRandomPoint() – повертає випадкову точку із списку.
  - addRemainingPointsToResult() – додає усі точки, що залишилися, до результуючого маршруту.
  - findNearestPointInList() – знаходить середь списку точок найближчу до переданої.
- PointServiceImpl – імплементація інтерфейсу PointService. Сигнатури та поведінка його методів були описані в інтерфейсі.
- RouteServiceImpl – імплементація інтерфейсу RouteService. Сигнатури та поведінка його методів вже були описані в інтерфейсі.

Таким чином, ми розглянули усі класи рівня бізнес-логіки та побачили, що кожен із них має свою цілком визначену та логічну роль, а також те, що їх можна легко перевикористовувати у іншому проекті за потреби.

3.5 Розробка REST-модуля

REST-модуль у даному проекті містить сутності, які приймають участь у організації інтерфейсу роботи із клієнтами даного сервісу, тобто, у організації API. Модуль має єдиний клас RouteSearchResource на першому рівні, а також три внутрішніх пакети : request, response та mapper, що містять об’єкти запитів, відповідей та мапери відповідно.

Головний клас даного модулю лежить на першому рівні :

- RouteSearchResource – головний клас модуля REST даної системи. Він представляє REST – контролер, який обробляє запити зі сторони клієнта, перевіряє їх на коректність, логує, викликає методи мапера для перетворення запитів у бізнес-сутності, викликає методи сервісів із бізнес-сутностями у разі аргументів, отримує відповіді від сервісів, знову мапить їх, але на цей раз вже у відповідь, загортає такі відповіді у об’єкт ResponseEntity та відправляє відповідь клієнтові. У ресурсі є такі endpoint-и :
  - ResponseEntity<RouteSearchResponse> findShortestRoute (@Valid @RequestBody RouteSearchRequest request) – метод, який приймає HTTP-запити із тілом-об'єктом RouteSearchRequest, в залежності від внутрішнього параметру timeCritical викликає метод того чи іншого інтерфейсу для обчислень із аргументом – перетвореним у ентіті запитом, а потім перетворений результат у відповідь відправляє назад клієнтові.
  - ResponseEntity<RouteSearchResponse> findShortestRouteViaGeneticAlgorithm (@Valid @RequestBody GeneticRouteSearchRequest request) – метод, який приймає HTTP-запит із тілом – об’єктом GeneticRouteSearchRequest та викликає для обчислень метод GeneticRouteSearchService із перетвореним запитом у якості аргумента, після цього повертаючи перетворений у відповідь результат, переданий сервісом.
  - ResponseEntity<RouteSearchResponse> findShortestRouteViaGreedyAlgorithm(@Valid @RequestBody GreedyRouteSearchRequest request) – метод, який приймає HTTP – запит із тілом – об’єктом GreedyRouteSearchRequest

та викликає для обчислень метод GreedyRouteSearchService із перетвореним записом у якості аргумента, після цього повертаючи перетворений у відповідь результат, що був переданий сервісом, клієнтові.

Також у даному модулі на других рівнях знаходяться такі класи, як :

- RouteSearchDataMapper – інтерфейс мапера, який містить чотири методи для перетворення сутностей у відповіді та запитів у сутності. Імплементація маперу створюється бібліотекою MapStruct на етапі компіляції програми.
- PointRequest – модель запиту, що представляє собою точку, і містить дві цілочисельні координати x та y усередині.
- RouteSearchRequest – модель запиту, що відображає собою запит для пошуку маршруту. Всередині знаходиться список точок, а також булевське значення важливості часу для клієнтського запиту.
- GeneticRouteSearchRequest – модель запиту, що відображає собою запит для пошуку маршруту за допомогою генетичного алгоритму. Дана модель запиту дозволяє окрім списку вхідних точок конфігурувати такі поля – параметри генетичного алгоритму, як вірогідність мутації, розмір турніру, розмір кожної популяції, кількість популяцій у еволюції, чи потрібно враховувати елітарність особин, а також скільки елітних особин буде у кожнеій популяції.
- GreedyRouteSearchRequest – модель запиту, що відображає запит для обчислення маршруту за допомогою алгоритму найближчого сусіда. Усередині на даний момент знаходиться тільки список вхідних точок для обчислення.
- PointResponse – модель відповіді, що відображає відповідь у вигляді точки як мінімального елементу відповіді. Тобто, точки не

повертаються як відповіді у даній системі, але вони є частиною кожної відповіді. Усередині два параметри – координати по x та y.

- RouteSearchResponse – модель відповіді, що відображає відповідь у вигляді маршруту, кількості мілісекунд, що були витрачені на обчислення, а також дистанцію вирахуваного маршруту.

Як бачимо, розглянуті у даному розділі класи REST модуля прекрасно виконують своє загальне призначення, а саме створення інтерфейсу для взаємодії з користувачем. Також вони чудово справляються із валідуванням вхідних запитів та забезпеченням безпеки системи, адже внутрішні бізнес-сутності ні приймаються напряму від клієнта, ні віддаються напряму клієнту - все це проходить через спеціальні об'єкти request і response відповідно.

3.6 Розробка допоміжних класів та конфігурації

У даній системі були розроблені три допоміжних класи :

- RouteSearchApplication – клас – вхідна точка для Spring Boot додатку, через яку власне й запускається додаток. Також даний клас за допомогою анотацій допомагає нам сконфігурувати використання properties-файлів.
- Constants – утилітний клас із константами, які винесені із інших допоміжних класів та ресурсів для покращення читабельності та редагованості.
- DocumentationConfig – клас, який використовується для конфігурації документування нашого API за допомогою Swagger.

Також у директорії resources знаходяться наступні допоміжні файли :

- Application.properties – це файл, у якому ми можемо конфігурувати фреймворкові змінні. В даному випадку ми конфігуруємо у ньому порт додатку та кодування вихідних файлів.

- Values.properties – це файл, у якому ми задаємо дефолтні значення певних змінних. У данному випадку ми конфігуруємо значення за замовчуванням для генетичного алгоритму.
- Logback.xml – це файл, у котрому ми задаємо конфігурацію для логування за допомогою логера LogBack. У даному випадку конфігурація спрямована на два різні виводи логів – у файл та на консоль.

Можна зробити підсумок, що усі розглянуті у даному розділі допоміжні сутності не просто так знаходяться у проекті та дуже добре виконують свої ролі, і, хоч ці файли і є допоміжними, проте без них цей проект би не працював.

ВИСНОВОК ДО РОЗДІЛУ 3

У даному розділі були наведені деталі розробки нашої системи.

Зокрема, на початку розділу були наведені основні архітектурні рішення у системі та показана схема, як ця система загалом має працювати, які запити отримувати, що при цьому робити, яка бізнес логіка приблизно має відбуватися, що має бути потім, та що власне врешті-решт має повернутися до клієнта, що відсилав запит, як результат.

Далі була розглянута ієрархія пакетів у розроблюваній системі, що побудована у відповідності до стандартів Maven проектів, що мають дотримуватися у кожному проекті, у якому він використовується.

Після цього ми розглянули окремо розробку кожного модуля. Для всіх модулей були пояснені структура модуля, основні його класи та інтерфейси, їх функції, а також описані усі публічні та приватні методи усіх цих класів та інтерфейсів. Також для кожного модуля були описані POJO, що знаходяться у ньому, або DTO.

Першим із модулей був розглянутий модуль бізнес-сутностей, який потрібен у системі задля того, щоб використовувати його сутності як POJO – класи, із якими як з даними працюватимуть сутності бізнес-логіки нашої системи.

Другим із модулей був розглянутий модуль бізнес-логіки, який є головний об’єктом уваги у нашій системі, адже у ньому відбуваються усі бізнес-процеси даного додатку, без винятку.

Далі був розглянутий REST-модуль, що є своєрідним посередником між нашою бізнес логікою та клієнтом. Даний модуль займається низкою обов’язків, а саме – прийманням запитів на свох ендпоїнтах, їх перевірка на коректність, логування, мапінг DTO у POJO, виклик методів сервісів, отримання відповідей із сервісів, знову мапінг тепер вже POJO у DTO, та врешті відправка HTTP-відповіді назад клієнтові.

У кінці розділу були розглянуті розроблювані допоміжні класи та конфігураційні файли, які не можна віднести до жодного із вище перерахованих, проте вони також є невід’ємними частинами системи, й без них вона не працюватиме, як потрібно.

РОЗДІЛ 4
АНАЛІЗ РОЗРОБЛЕНОЇ СИСТЕМИ

У даному розділі розглядаються такі теми, як демонстрація розробленої документації API, демонстрація роботи програми, інтеграційне тестування системи, а також рекомендації щодо подальшого розвитку додатка.

4.1 Підготовка до демонстрації роботи програми

Розроблений застосунок у даній роботі представляє із себе REST API, яке може використовуватися будь-яким клієнтом за допомогою HTTP-запитів на відповідні ендпоїнти програми.

Для того, щоб продемонструвати роботу програми, ми використаємо такий клієнт для виконання HTTP-запитів, як Postman, інтерфейс якого зображений на рис. 4.1. Він нам дозволить робити запити по HTTP та отримувати відповіді на них від наших ендпоїнтів.

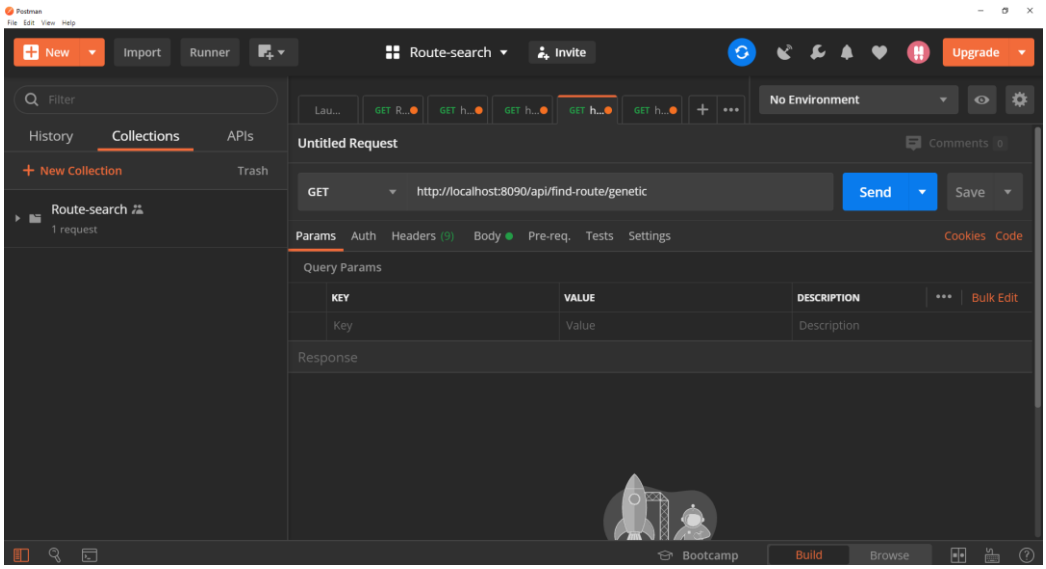


Рис 4.1. Інтерфейс Postman

Для того, щоб запустити в роботу наш додаток, нам необхідно спочатку скомпілювати написаний програмний код. Це ми можемо зробити за допомогою засобів нашої IDE – Intellij IDEA 2019.3.4 (Ultimate Edition). Для цього буде використаний Java Development Kid версії 9, завантажений із сайту Oracle. Після того, як ми скомпілюємо наш код, програма запуститься і буде

доступна для запитів за адресою localhost:8090/api, який був сконфігурований у файлі application.properties.

4.2 Демонстрація розробленої документації API

Запустивши програму, ми можемо перейти до першого пункту огляду її роботи, а саме до розгляду документації. Дана документація розроблена із допомогою Swagger та має дві версії. Розглянемо обидві із них.

Перший і дуже важливий вид документації – це JSON – представлена інформація про наш API. Для її отримання по HTTP застосуємо Postman. Необхідний запит зображено на рис. 4.2.

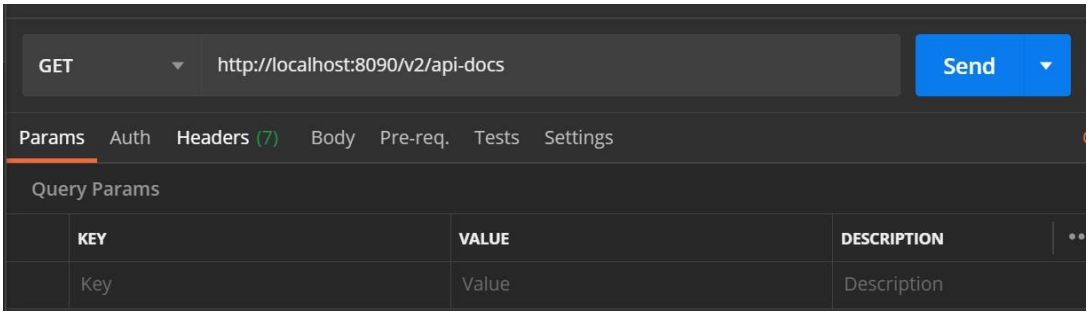


Рис 4.2. Запит для отримання JSON документації

Як бачимо, для отримання даного типу документації нам знадобилося лише відправити GET-запит на адресу localhost:8090/v2/api-docs. Результат, отриманий від нашого API, можемо побачити на рис. 4.3.

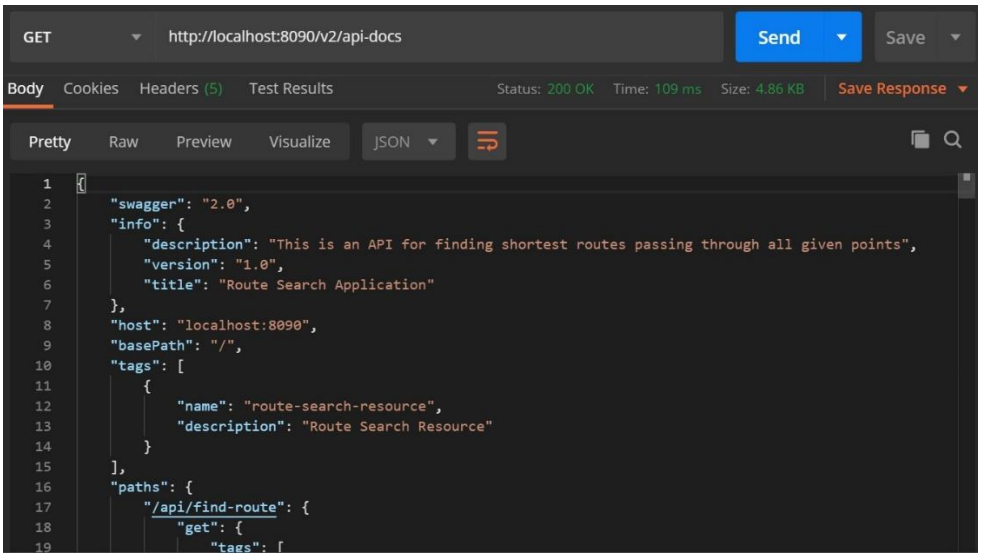


Рис 4.3. Відповідь із JSON-документацією

Як бачимо, даний запит повернув нам повну інформацію про API, загальний опис додатку, усі ендпоїнти, а також усі стандарти запитів та відповідей для даних ендпоїнтів.

Тепер розглянемо інший вид документації. Він є графічним, і отриманий може бути у двох виглядах – все таким же чином – через Postman, що може використовуватися клієнтами для отримання HTML-сторінки та її передачі на відрисовку. Відповідний запит із відповіддю на нього показані на рис. 4.4.

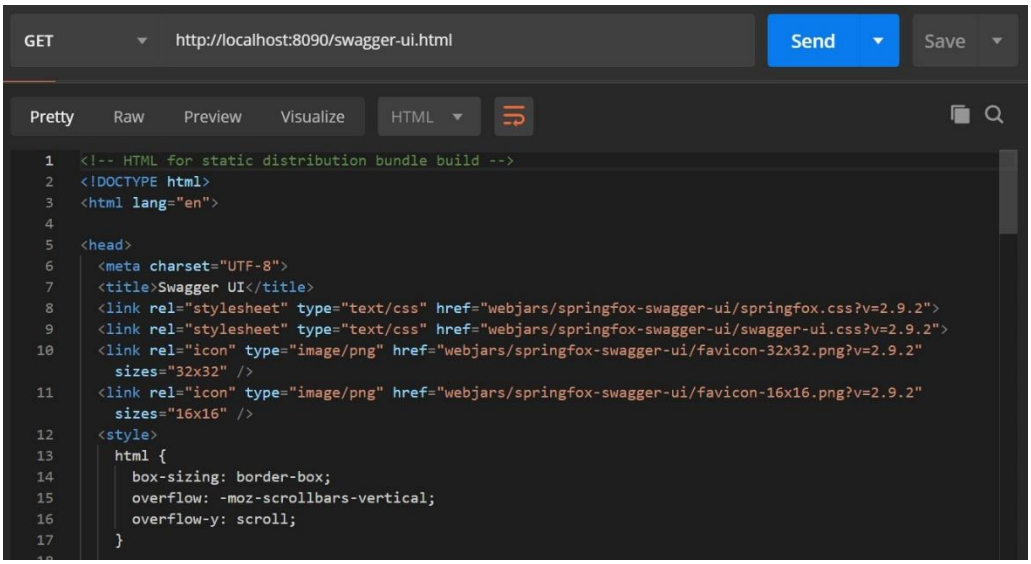


Рис 4.4. Отримання HTML-документації

Як бачимо із даного скріншоту, за запитом на адресу localhost:8090/swagger-ui.html ми отримали розмітку HTML із документацією.

Тепер отримаємо документацію із використанням браузера (рис 4.5).

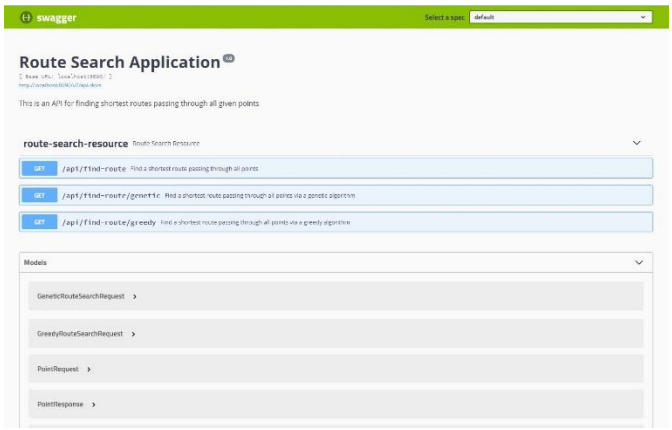


Рис 4.5. Отримання HTML-документації

Як бачимо, теперь документація дійсно добре виглядає та є дуже читабельною.

Розглянемо отриману документацію більш докладно. На рис. 4.6. зображена документація одного із ендпоїнтів.

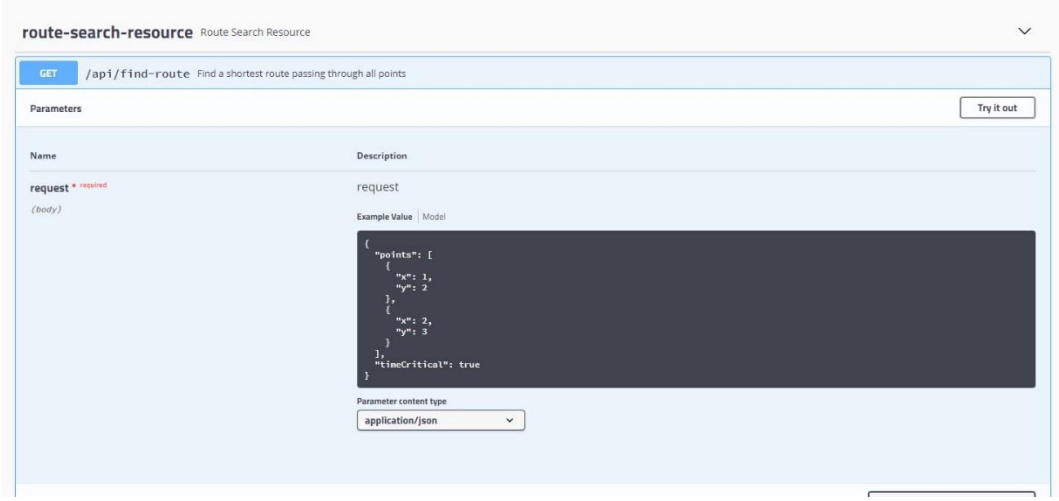


Рис 4.6. Документація ендпоїнта

Як бачимо, для конкретно взятого ендпоїнта задокументовані можливі HTTP-операції (у даному випадку тільки GET), також надана інформація щодо структури запиту, що дозволена для передавання на цей ендпоїнт, наданий її приклад, а також зазначено, яку відповідь повертатиме такий запит.

На рис. 4.7. зображена документація структури одного із запитів.

Рис 4.7. Документація структури запиту

4.3 Демонстрація роботи додатку

Розглянемо роботу нашого розробленого API та переконаємося, що всі ендпоїнти працюють так, як ми і очікуємо. Для цього застосуємо Postman.

Для початку, спробуємо відправити запит на перший із ендпоїнтів, без додавання тіла запиту. Даний запит зображений на рис. 4.8.

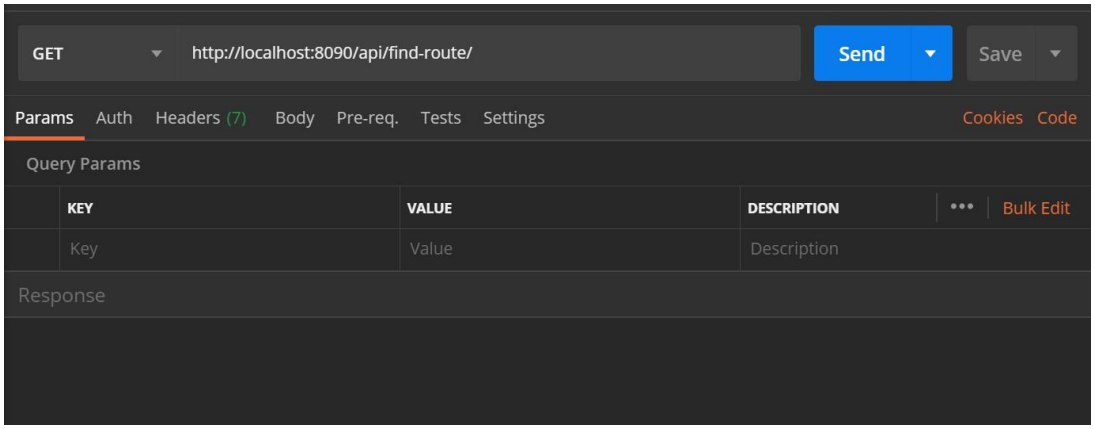


Рис 4.8. Запит без тіла на перший ендпоїнт

У результаті даного запиту отримаємо наступну відповідь (рис. 4.9).

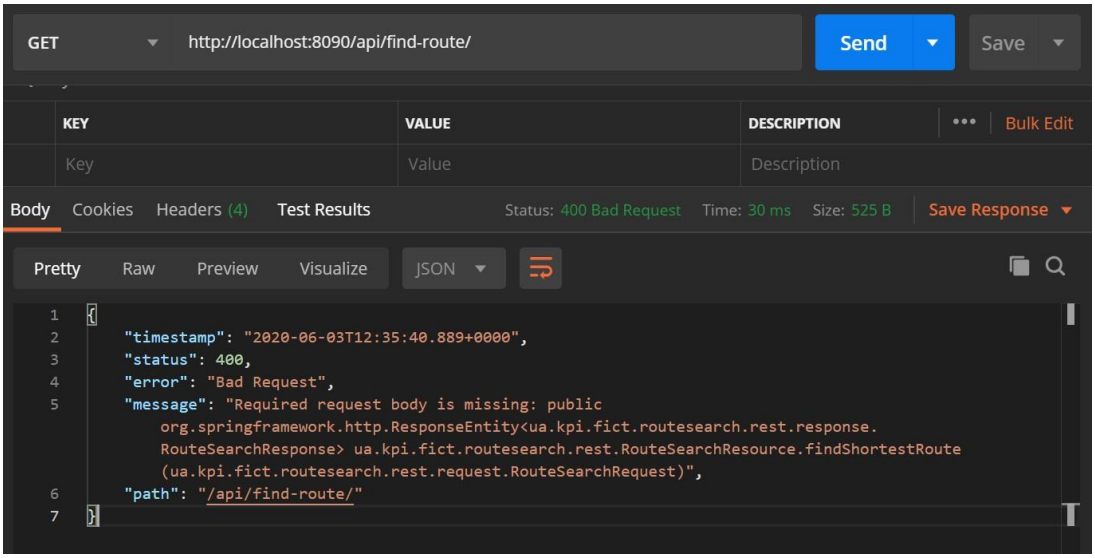


Рис 4.9. Відповідь на запит із пустим тілом від першого ендпоїнту

Як бачимо, сервер відправив нам відповідь із статус-кодом 400, що означає те, що відправлений нами запит був некоректним. Ця поведінка саме така, як ми й очікували, оскільки ми просимо сервер вирахувати маршрут, при цьому не передавши точки, між якими й необхідно його будувати.

Виправимо це, та зробимо запит на той же ендпоїнт, але на цей раз передавши список точок, а також вкажемо, що час для нас не принциповий. Даний запит зображений на рис. 4.10.

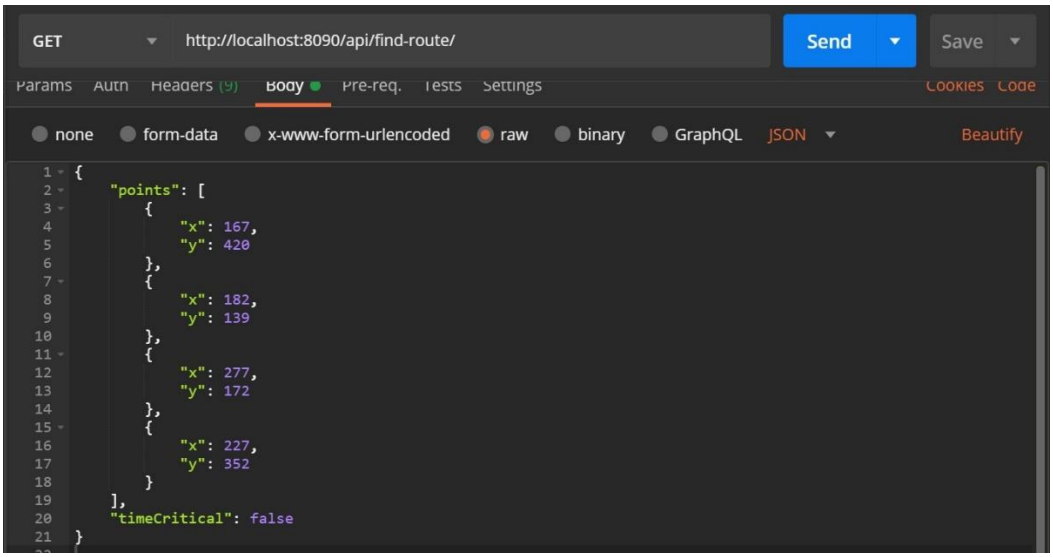


Рис 4.10. Запит на перший ендпоїнт із списком точок

Оскільки ми вказали, що час не важливий для нас у даному випадку, то обчислення будуть зроблені за допомогою генетичного алгоритму. Відповідь на даний запит зображена на рис. 4.11.

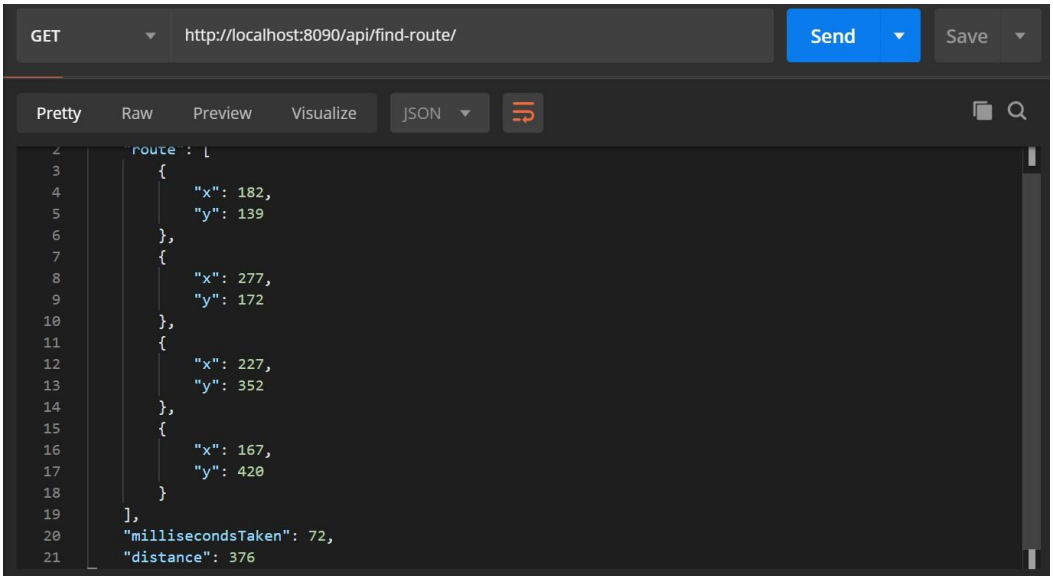


Рис 4.11. Відповідь від першого ендпоїнта на запит із неважливістю часу

Як бачимо, маршрут між точками був прорахований і повернутий за 72 мілісекунди і склав 376 умовних одиниць.

Теперь перевіримо, що зробить аналогічний запит на той же ендпоїнт, але на цей раз із вказанням, що час розрахунків для нас є критичним. Даний запит зображений на рис. 4.12.

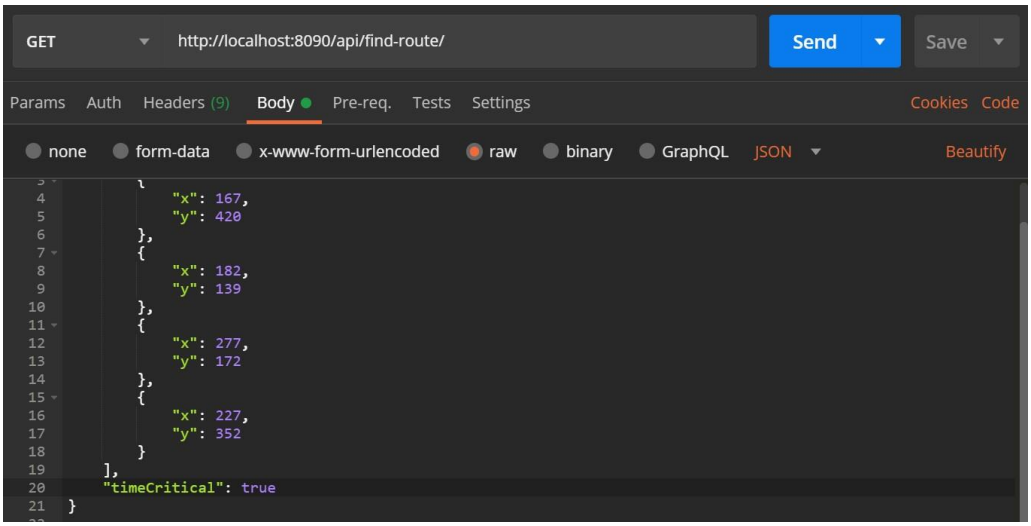


Рис 4.12. Запит на перший ендпоїнт із вказанням важливості часу

Як бачимо, на цей раз ми виставили параметр запиту timeCritical у true. Можемо побачити на рис. 4.13 відповідь сервера на цей запит.

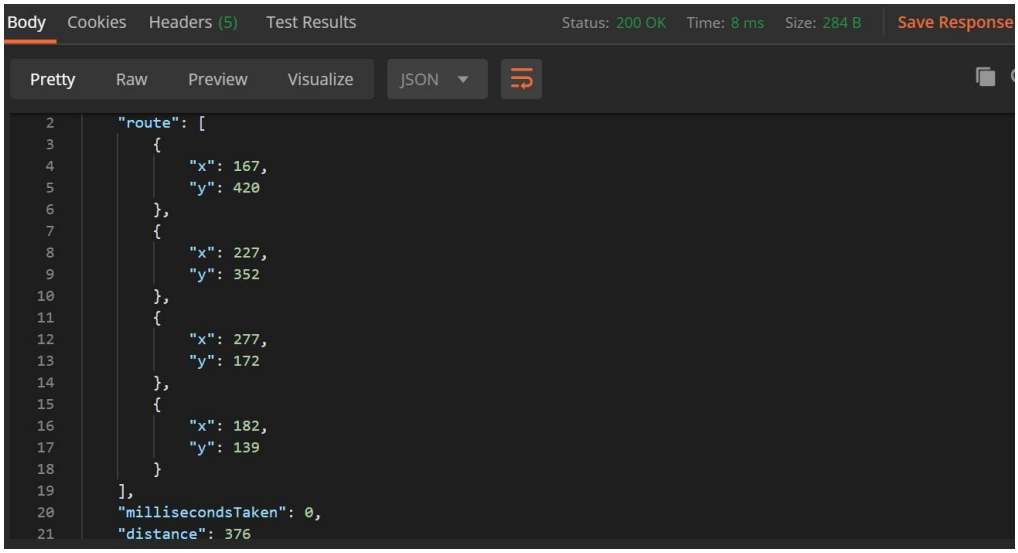


Рис 4.13. Відповідь від першого ендпоїнта на запит із важливістю часу

Як бачимо, теперь вже час обчислення становить менше однієї мілісекунди, це через те, що на цей раз розрахунки були виповнені за допомогою жадібного алгоритму. Дистанція ж зосталася точно такою, але це тільки через те, що точок у вхідній умові було передано досить мало.

Тепер випробуємо другий ендпоїнт, що знаходиться за адресою /genetic відносно адреси першого ендпоїнта та який виконує обчислення за допомогою генетичного алгоритму. Для цього уведемо нове значення адреси. При цьому вкажемо для початку тільки список точок для обчислення. Запит зображений на рис. 4.14.

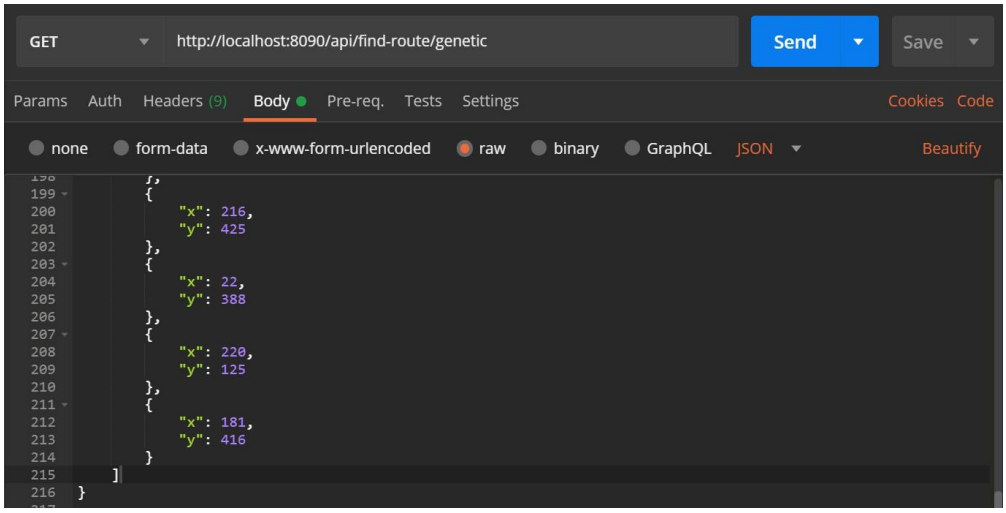


Рис 4.14. Запит на другий ендпоїнт

Як і у інших запитах до цього, ми вказали у його тілі список точок у форматі JSON, попередньо для цього вибравши опції «Body», та потім «raw».

Відповідь на даний запит зображена на рис. 4.15.

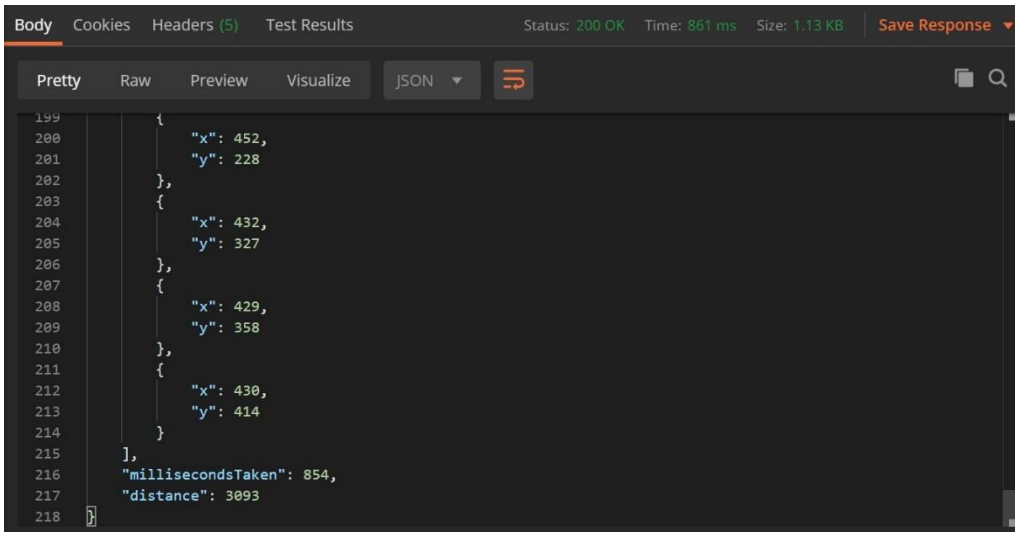


Рис 4.15. Відповідь на запит до другого ендпоїнту

Як бачимо, на цей раз кількість точок у запиті була збільшена до п’ятидесяти, тому обчислення зайняло набагато більше часу, враховуючи

специфіку генетичного алгоритму. Тим не менш, результат чудовий – 3093 умовних одиниці близький до якнайоптимальнішого у даному випадку.

Тепер спробуємо сконфігурувати власні параметри генетичного алгоритму у запиті. Такий запит показаний на рис. 4.16.

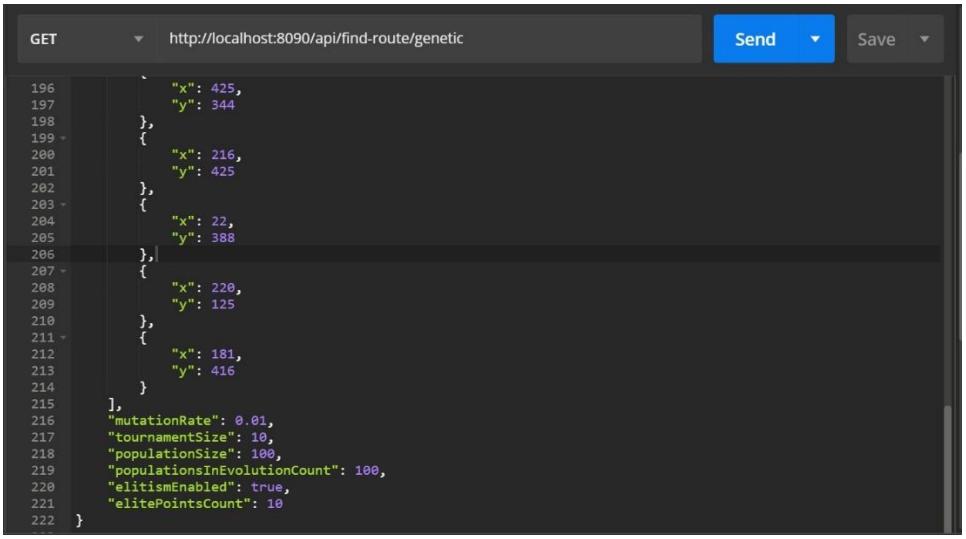


Рис 4.16. Запит до другого ендпоїнту із генетичними параметрами

Як бачимо, у даному запиті ми передали свої генетичні параметри, такі як шанс мутації, розмір турніру, розмір популяції, кількість популяцій у еволюції, показник включеності елітарності, а також кількість елітних точок. Відповідь на даний запит зображена на рис. 4.17.

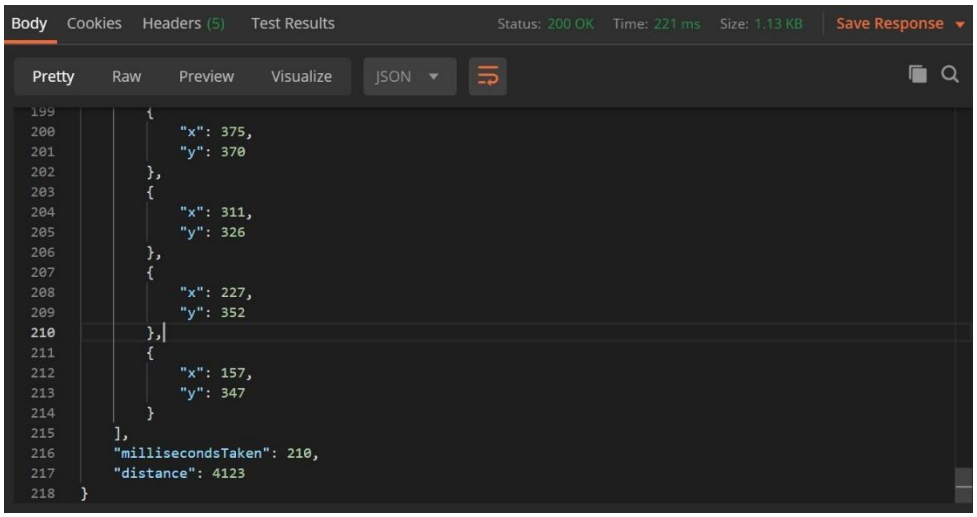


Рис 4.17. Відповідь на запит із генетичними параметрами

Таким чином, теперь час, який зайняли обчислення, склав 210 мілісекунд, тоді як загальна довжина маршруту – 4123 умовних одиниць. Як бачимо, хоч

довжина шляху через зміну параметрів генетичного алгоритму у запиті збільшилася на третину, тим не менш ми досягнули прискорення розрахунків аж у 4 рази.

Останнім із ендпоїнтів ми подивимось на роботу третього, який виконує розрахунки за допомогою алгоритму найближчого сусіда. Його адреса – це localhost:8090/api/find-route/greedy. Він приймає запити тільки із списком точок у тілі, та відповідатиме із кодом BadRequest на усі інші типи запитів, надіслані на його адресу..

Запит до нього зображений на рис. 4.18.

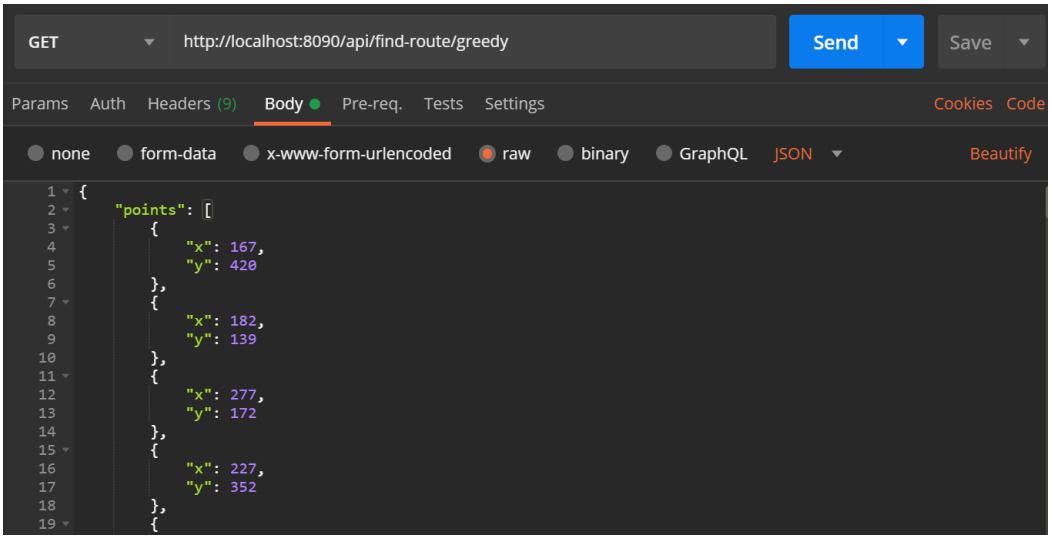


Рис 4.18. Запит до третього ендпоїнта

Як бачимо, все, що потрібно у тілі запиту до ендпоїнта жадібного алгоритма – це список точок. Адже даний алгоритм є доволі простим та не потребує додаткових параметрів. Також варто зазначити, що даний запит ми робимо із тією ж кількістю та врешті із тими ж самими точками, що й до цього робили до другого ендпоїнта, із метою порівняти час роботи та вирахувану довжину маршруту за допомогою двох алгоритмів.

Результат, переданий від сервера, зображений на рис. 4.19.

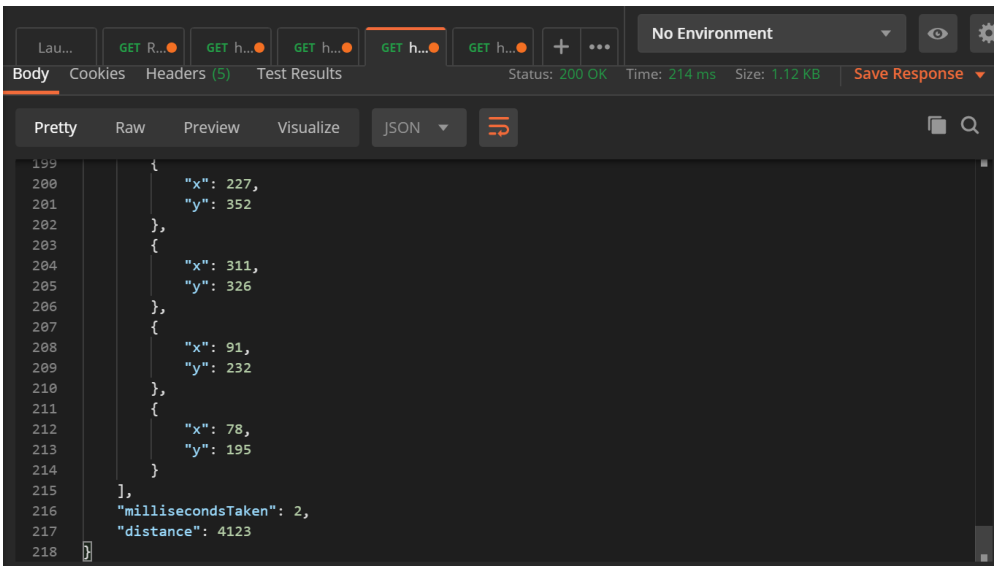


Рис 4.19. Відповідь на запит до третього ендпоїнта

Як бачимо, маршрут для тих же самих точок знайдений за допомогою алгоритму найближчого сусіда доволі сильно відрізняється від відповідно знайденого за допомогою генетичного алгоритму. Помітно, що обчислення виконуються набагато швидше, але при цьому довжина шляху теж є суттєво більшою, тобто шлях є менш оптимальним.

4.4 Інтеграційне тестування додатку

Для проведення автоматичного тестування усіх можливих випадків у роботі із нашим розробленим додатком були розроблені тест-кейси, усі з яких перераховані у таб. 1.

Таблиця 1. Тест-кейси інтеграційного тестування

№ п/п	Опис	Очікувані результати
1	Запит до /find-route із коректним списком точок та timeCritical = false у тілі	Побудований маршрут, його довжина та кількість витраченого на обчислення часу у відповіді
2	Запит до /find-route із коректним списком точок та timeCritical = true у тілі	Побудований маршрут, його довжина та кількість витраченого на обчислення часу у відповіді

Продовження таблиці 1. Тест-кейси інтеграційного тестування		
3	Запит до /find-route без списка точок	Відповідь зі статусом BadRequest
4	Запит до /find-route/genetic із коректним списком точок, але без генетичних параметрів	Побудований маршрут, його довжина та кількість витраченого на обчислення часу у відповіді
5	Запит до /find-route/genetic із коректним списком точок та з усіма генетичними параметрами	Побудований маршрут, його довжина та кількість витраченого на обчислення часу у відповіді
6	Запит до /find-route/genetic без списку точок	Відповідь зі статусом BadRequest
7	Запит до /find-route/genetic із коректним списком точок та з усіма генетичними параметрами, кількість елітних точок передана рівна 0	Відповідь зі статусом BadRequest
8	Запит до /find-route/greedy із коректним списком точок	Побудований маршрут, його довжина та кількість витраченого на обчислення часу у відповіді
9	Запит до /find-route/genetic без списку точок	Відповідь зі статусом BadRequest

Запустимо за допомогою Intellij IDEA усі написані інтеграційні тести. Результат їх виконання зображений на рис. 4.20.

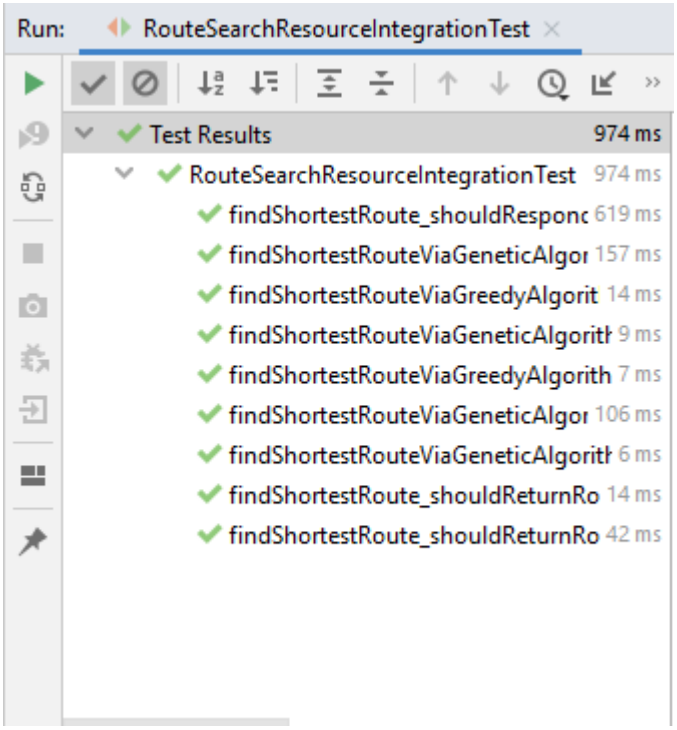


Рис 4.20. Результат виконання інтеграційних тестів

Як бачимо, усі інтеграційні тести були виконані успішно, тож наша програма працює саме так, як і очікувалося. Також варто зазначити, що дані тести можна буде використовувати як регресивні – при змінах у логіці програми можна буде перевіряти, чи вона все ще правильно працює.

4.5 Рекомендації щодо розвитку та вдосконалення додатка

У подальшому для покращення та поглиблення функціональності можна внести декілька напрямків змін. По-перше, можна додати опцію пошуку маршрутів у різних видах топологій. По-друге, можна додати опцію пошуку не тільки маршруту через усі точки, але й наприклад від однієї до іншої точки, що буде підтримуватися залежно від типу топології. Також ще одним напрямком розширення функціональності додатка може стати збільшення кількості алгоритмів, використовуваних для обчислень. Якщо ж говорити про розширення із точки зору технологій, можна, наприклад, додати можливість роботи із новими протоколами та обміну за допомогою них даними із клієнтами. Наприклад, роботу із XML або BSON. У будь-якому разі, варто зазначити, що завдяки тому, що маються інтеграційні тести, буде легко

вводити нові функції, при цьому точно впевнюючись, що старі не ламатимуться.

ВИСНОВОК ДО РОЗДІЛУ 4

У даному розділі був представлений готовий додаток, що був розроблений за усе виконання дипломної роботи. Була надана загальна інструкція користування API додатка, а також були розглянуті й випадки неправильного користування і до яких ситуацій вони можуть використовуватися.

Була розглянута документація даного REST API, яка представлена у ньому у двох видах і може отримуватися, наприклад, для клієнтів, що споживають JSON, а також і для клієнтів, що бажають продивитися та вивчити документацію додатка через браузер.

Також була розглянута робота загалом із API додатка та із усіма його ендпоїнтами. Було показано призначення кожного із них, а також були продемонстровані приклади запитів на ці ендпоїнти та відповіді, що повертаються із них.

Далі були наведені тест-кейси інтеграційного тестування, що є дуже важливим для подібних додатків, адже потім саме воно чудово служитиме у вигляді регресивних тестів, які при додаванні нового функціоналу чи внесенні якихось змін до того, що є, будуть давати нам впевненість у тому, що бізне-логіка все ще працює, як потрібно.

Наприкінці розділу були наведені можливі ідеї для розширення можливостей додатка та покращення його функціоналу, які цілком можуть бути втіленими у життя у подальшому.

ВИСНОВКИ

Під час виконання даної дипломної роботи було створено систему для пошуку маршрутів у топологіях на основі еволюційних алгоритмів. Дана система допомагає своїм користувачам за допомогою HTTP-запитів на її API знаходити маршрути у повнозв’язних топологіях.

У першому розділі спочатку були детально розглянуті основні на сьогоднішній день існуючі види топологій, такі як гіперкуб, зірка, кільце, дерево, повнозв’язна, решітчата та лінійна топології. Було встановлено їх переваги та недоліки, а також розглянуті були підходи до маршрутизації у цих топологіях. Далі у цьому розділі були розглянуті різні види евристичних алгоритмів, які активно використовуються для вирішення проблеми знаходження маршрутів у топологіях. Було розглянуто основні принципи, а також і детальні покрокові алгоритми.

У другому розділі даної роботи були розглянуті технології, що використалися для розробки нашої системи. Були наведені їх детальні переваги та об обґрунтування доцільності їх вибору. Було встановлено, що кожна із обраних технологій дійсно дуже допоможе скоротити строки розробки, спростити її для розуміння, а також спростити її подальшу підтримку.

У третьому розділі були розглянуті деталі розробки нашої системи. Зокрема, була показана загальна спроектована архітектура системи, пояснена побудова директорно-пакетної системи. Після цього у даному розділі були окремо розглянуті модулі та їх розроблені компоненти. Першим із модулей став модуль бізнес сутностей, що є дуже важливим, оскільки саме його сутності використовуються для зберігання бізнес-інформації. Другим був розглянутий найважливіший модуль данох системи, а саме модуль бізнес-логіки. Далі ми розглянули REST-модуль, важливість якого важко недооцінити, адже саме у ньому знаходиться уся специфічна для даної платформи логіка, а також усі заходи щодо безпеки також на себе бере цей модуль. Наостанок у даному розділі були розглянуті допоміжні класи та конфігурації системи, які не входять

до жодного з перерахованих модулей, ате тим не менш є досить важливими для нашої системи.

У четвертому розділі ми провели детальний аналіз розробленої системи. Було проведено підготовку до запуску та демонстрації працюючого додатку. Після цього було показано, як будь-який користувач власне перед використанням розробленої системи може подивитися її документації у зручному для нього вигляді, аби зрозуміти, як потрібно взаємодіяти із даною системою та що ця взаємодія зможе йому принести. Далі була продемонстрована робота програми із точки зору користувача, який відправляє запити до неї. Були показані як позитивні, так і негативні кейси димового тестування. Після цього були продемонстровані у дії інтеграційні тести для даної системи, що покривають усі можливі випадки користуванням даним розробленим API. У якості підсумку були дані рекомендації щодо подальших можливих напрямків розвитку додатку.

Слід також зазначити, що під час розробки програми були враховані усі зазначені на початку умови, яким вона мала відповідати.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Цилькер, Б., Орлов, С. (2011). Організація комп'ютерів та систем: Книга для вищих навчальних закладів -2га версія. (ст. 688).

2. Курейчик, В. (1998). Генетичні алгоритми – 2га версія (ст. 5 – 11).

3. Курейчик, В., Кажаров, А. (2008). Про деякі модифікації мурашиного алгоритму – 4та версія (ст. 7-11).

4. Левітін, А. (2006). Введення в дизайн & Аналіз алгоритмів (ст. 141).

5. Новосибірський державний університет – [Електронний ресурс]. Режим доступу : http://www.nsc.ru/win/elbib/data/show_page.dhtml?77+791 (дата запиту – 11.05.2020) – Топологія гіперкубу.

6. Авторська зона – [Електронний ресурс]. Режим доступу : http://author.nbpublish.com/kp/article_11380.html (дата запиту – 11.05.2020) – Опис базових топологій за допомогою графів у цілях побудови нотації комп'ютерних мереж.

7. BrestProg – [Електронний ресурс]. Режим доступу : https://brestprog.by/topics/heap/ (дата запиту – 11.05.2020) – Повне бінарне дерево. Куча. Черга із пріоритетом.

8. Evileg – [Електронний ресурс]. Режим доступу : https://evileg.com/ru/post/34/ (дата запиту – 11.05.2020) – Топології мереж передачі даних

9. Новосибірський державний університет – [Електронний ресурс]. Режим доступу:http://www.nsc.ru/win/elbib/data/show_page.dhtml?77+790 (дата запиту – 11.05.2020) – Повнозв'язна топологія.

10. Комп'ютерні технології – [Електронний ресурс]. Режим доступу : https://sites.google.com/site/informtexxim/home/5 (дата запиту – 11.05.2020) – Топології мереж.

11. Хабр – [Електронний ресурс]. Режим доступу : https://habr.com/ru/post/105302/ (дата запиту – 11.05.2020) – Мурашині алгоритми.
12. Хабр – [Електронний ресурс]. Режим доступу : https://habr.com/ru/post/104055/ (дата запиту – 11.05.2020) – Природні алгоритми. Алгоритм поведінки рою бджіл.
13. Хабр – [Електронний ресурс]. Режим доступу : https://habr.com/ru/post/128704/ (дата запиту – 11.05.2020) – Генетичний алгоритм. Просто про важке.
14. Medium – [Електронний ресурс]. Режим доступу : https://medium.com/nuances-of-programming/плюсы-и-минусы-программирования-на-java-2861f4c2a0d5 (дата запиту – 16.05.2020) – Плюси та мінуси програмування на Java.
15. ProgLang – [Електронний ресурс]. Режим доступу : http://proglang.su/java/introduction-to-programming#istoriya-sozdaniya-yazyka-java (дата запиту – 16.05.2020) – Огляд мови програмування Java
16. Javarush – [Електронний ресурс]. Режим доступу : https://javarush.ru/groups/posts/510-preimujshestva-ispoljhzovanija-spring (дата запиту – 16.05.2020) – Переваги використання Spring
17. ItProger – [Електронний ресурс]. Режим доступу : https://tproger.ru/articles/maven-short-intro/ (дата запиту – 16.05.2020) – Коротке знайомство із Maven
18. Хабр – [Електронний ресурс]. Режим доступу : https://habr.com/ru/post/435144/ (дата запиту – 16.05.2020) – Введення в Spring Boot. Створення простого REST API на Java.
19. Proselyte – [Електронний ресурс]. Режим доступу : https://proselyte.net/tutorials/junit/introduction/ (дата запиту – 16.05.2020) – Керівництво по JUnit. Введення.

20. Хабр – [Електронний ресурс]. Режим доступу : https://habr.com/ru/post/434798/ (дата запиту – 16.05.2020) – Swagger – розумна документація.
21. Хабр – [Електронний ресурс]. Режим доступу : https://habr.com/ru/post/438870/ (дата запиту – 16.05.2020) – Lombok повертає велич Java.
22. Codeflow – [Електронний ресурс]. Режим доступу : https://www.codeflow.site/ru/article/mapstruct (дата запиту – 16.05.2020) – Коротке керівництво по MapStruct.
23. Bauman National Library – [Електронний ресурс]. Режим доступу : https://ru.bmstu.wiki/index.php?title=Spring_Framework&mobileaction=toggle_view_desktop (дата запиту – 16.05.2020) – Spring Framework.
24. Tutorials Point – [Електронний ресурс]. Режим доступу : https://www.tutorialspoint.com/spring_boot/spring_boot_introduction.htm(дата запиту – 16.05.2020) – Spring Boot Introduction.
25. Java Study – [Електронний ресурс]. Режим доступу : http://javastudy.ru/junit/junit-hello-world/ (дата запиту – 16.05.2020) – JUnit – введення в юніт-тести.

ДОДАТОК 1

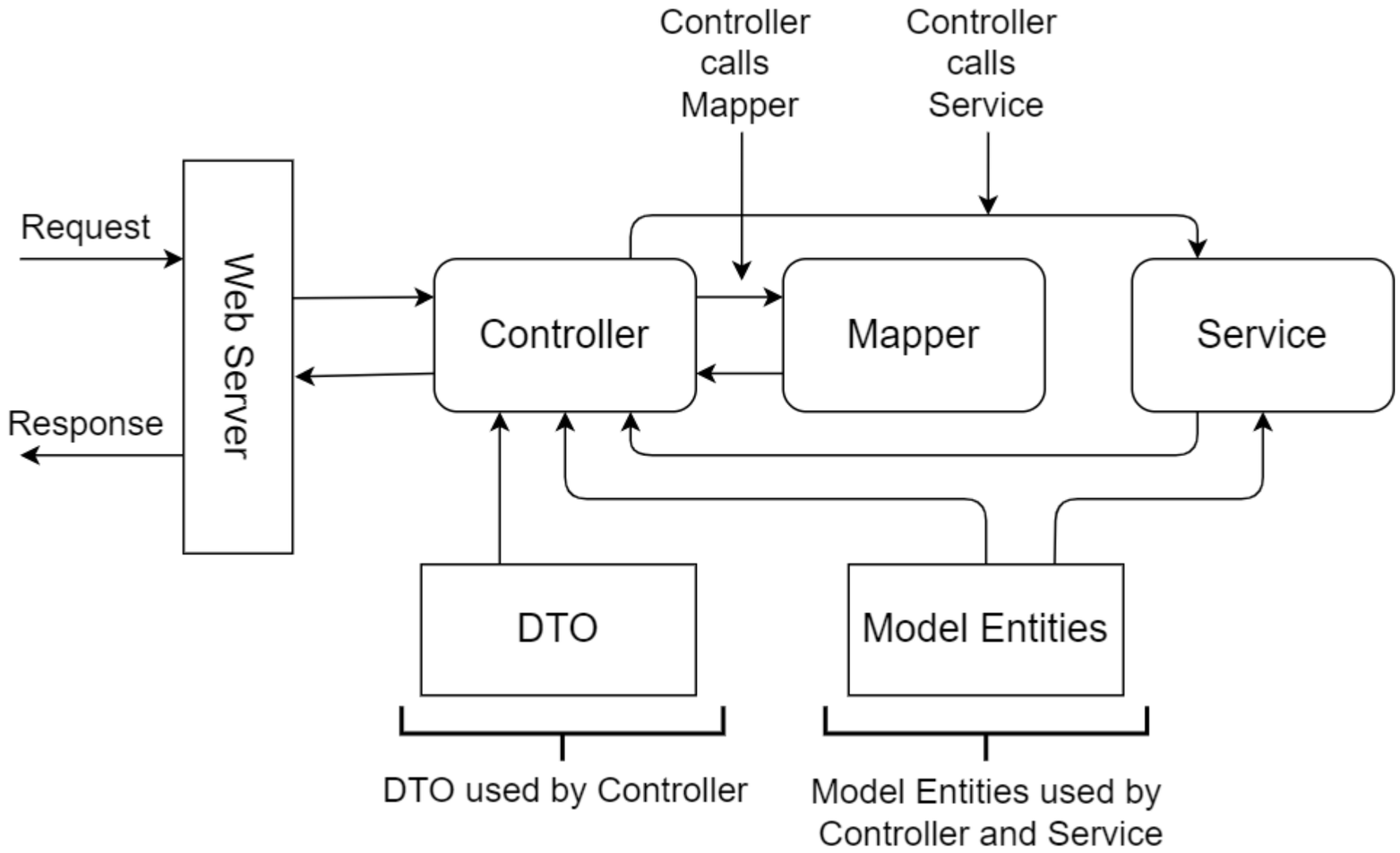
Система пошуку маршрутів в топологіях на основі еволюційних алгоритмів

Структурна схема системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2020 р



					ІАЛЦ.467200.004 Д1		
					Система пошуку маршрутів в топологіях на основі еволюційних алгоритмів Структурна схема системи	Літера	Маса / Масштаб
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Морозов А.С.				Аркуш 1	Аркушів 1
Перевірив		Волокита А.М.					
Н.контр.		Сімоненко В.П.			Дипломна робота	НТУУ "КПІ Ім Ігоря Сікорського", ФІОТ, гр. ІП-64	
Затверд.							

ДОДАТОК 2

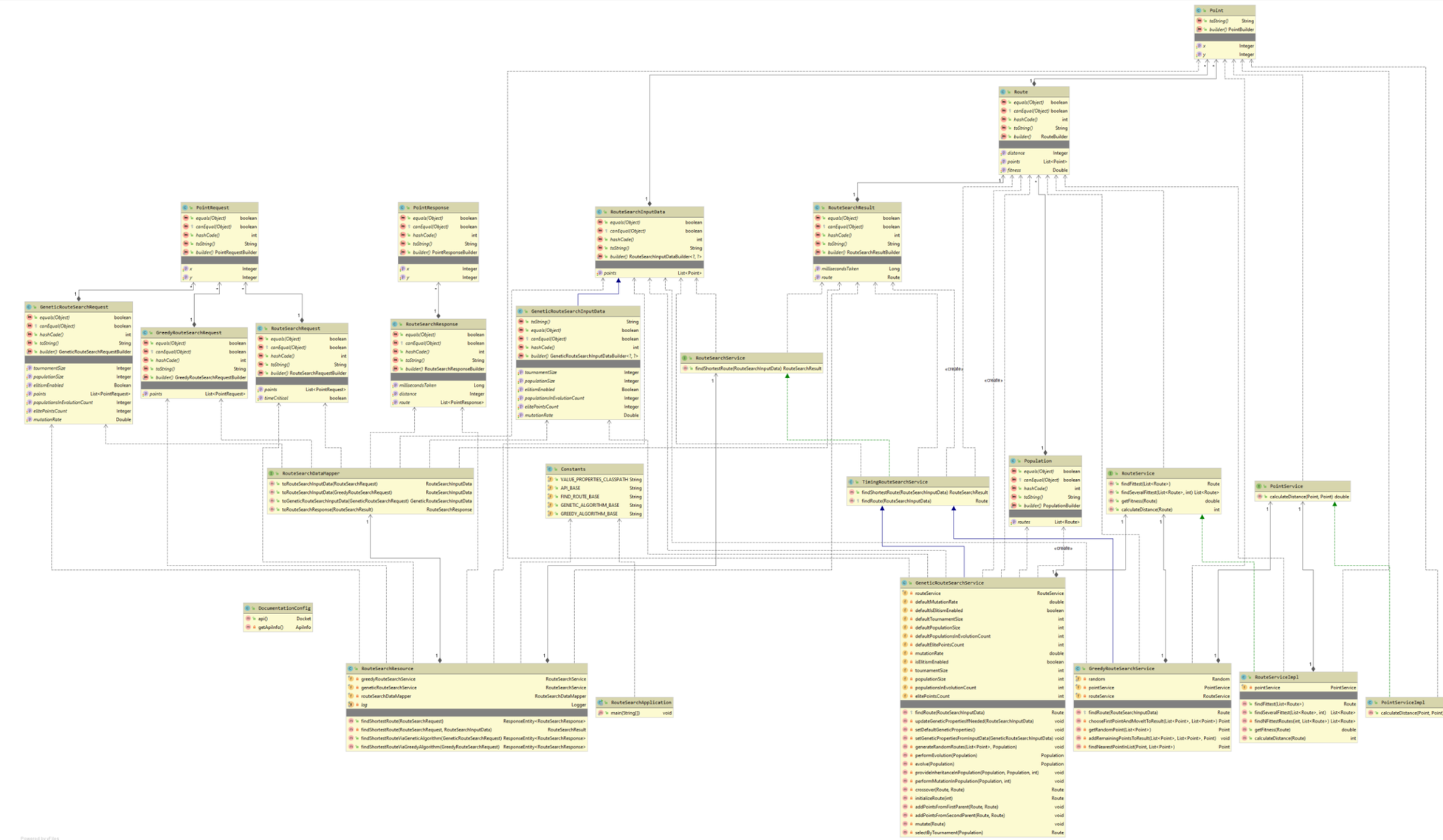
Система пошуку маршрутів в топологіях на основі еволюційних алгоритмів

Функціональна схема (діаграма класів)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2020 р



					ІАЛЦ.467200.005 Д2	
					Система пошуку маршрутів в топологіях на основі еволюційних алгоритмів Функціональна схема (діаграма класів)	Літера / Маса / Масштаб
Зм.	Арк.	№ докум.	Підпис	Дата		
Розробив		Морозов А.С.				Аркуш 1 / Аркушів 1
Перевірив		Волокита А.М.				
Н.контр.		Сімоненко В.П.			Дипломна робота	НТУУ "КПІ Ім Ігоря Сікорського", ФІОТ, гр. ІП-64
Затверд.						

ДОДАТОК З

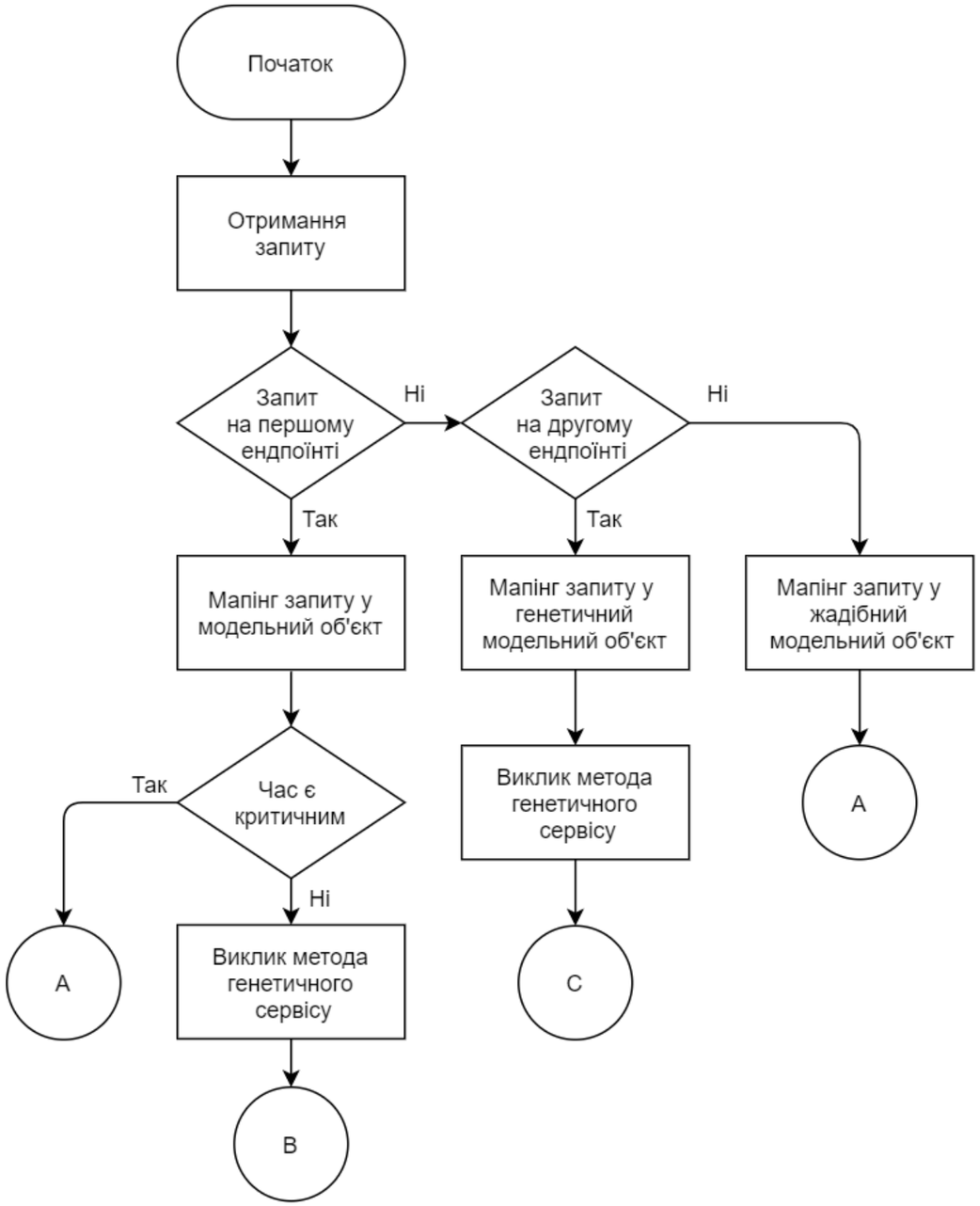
Система пошуку маршрутів в топологіях на основі еволюційних алгоритмів

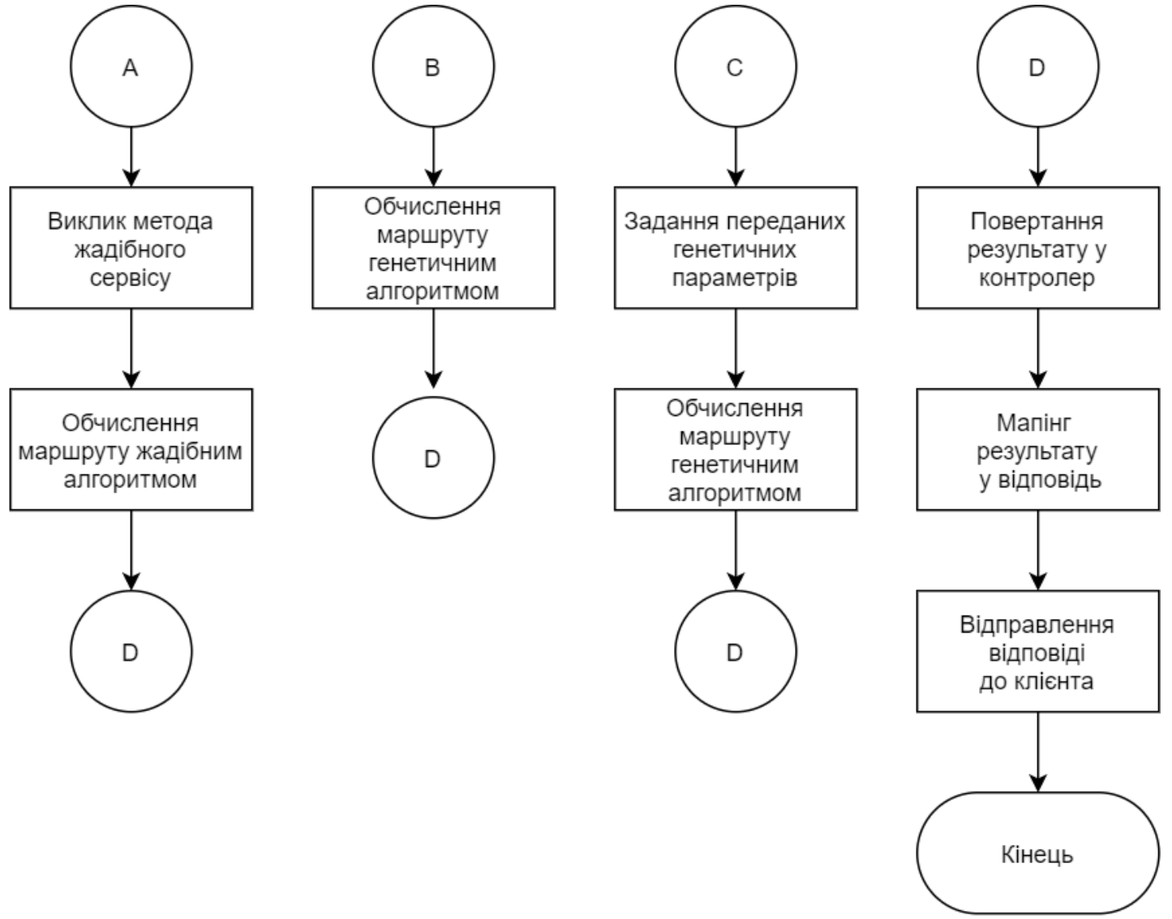
Алгоритм дій програмного забезпечення

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2020 р





					ІАЛЦ.467200.006 ДЗ			
					Система пошуку маршрутів в топологіях на основі еволюційних алгоритмів Алгоритм дій програмного забезпечення	Літера	Маса	Масштаб
Зм.	Арк.	№ докум.	Підпис	Дата				
Розробив		Морозов А.С.				Аркуш 1	Аркушів 1	
Перевірив		Волокита А.М.						
.								
Н.контр.		Сімоненко В.П.			Дипломна робота	НТУУ "КПІ Ім Ігоря Сікорського", ФІОТ, гр. ІП-64		
Затверд.								

ДОДАТОК 4

Система пошуку маршрутів в топологіях на основі еволюційних алгоритмів

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 13

Київ 2020 р

```java
package ua.kpi.fict.routesearch.service;

import ua.kpi.fict.routesearch.entity.RouteSearchInputData;
import ua.kpi.fict.routesearch.entity.RouteSearchResult;

public interface RouteSearchService {

    RouteSearchResult findShortestRoute(RouteSearchInputData inputData);
}
```

```java
package ua.kpi.fict.routesearch.service;

import java.time.Clock;

import ua.kpi.fict.routesearch.entity.Route;
import ua.kpi.fict.routesearch.entity.RouteSearchInputData;
import ua.kpi.fict.routesearch.entity.RouteSearchResult;

public abstract class TimingRouteSearchService implements RouteSearchService {

    @Override
    public RouteSearchResult findShortestRoute(RouteSearchInputData inputData) {
        Clock clock = Clock.systemDefaultZone();
        RouteSearchResult result = new RouteSearchResult();

        long millisecondsAtStart = clock.millis();
        Route route = findRoute(inputData);
        long millisecondsAtEnd = clock.millis();

        result.setRoute(route);
        result.setMillisecondsTaken(millisecondsAtEnd - millisecondsAtStart);
        return result;
    }

    protected abstract Route findRoute(RouteSearchInputData inputData);
}
```

```java
package ua.kpi.fict.routesearch.service;

import ua.kpi.fict.routesearch.entity.Point;

public interface PointService {

    double calculateDistance(Point firstPoint, Point secondPoint);
}
```

```java
package ua.kpi.fict.routesearch.service;

import java.util.List;

import ua.kpi.fict.routesearch.entity.Route;

public interface RouteService {

    Route findFittest(List<Route> routes);

    List<Route> findSeveralFittest(List<Route> routes, int quantity);
```

```
    double getFitness(Route route);

    int calculateDistance(Route route);
}
```

```
package ua.kpi.fict.routesearch.service.impl;

import static java.util.Objects.isNull;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import lombok.RequiredArgsConstructor;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

import ua.kpi.fict.routesearch.entity.GeneticRouteSearchInputData;
import ua.kpi.fict.routesearch.entity.Point;
import ua.kpi.fict.routesearch.entity.Population;
import ua.kpi.fict.routesearch.entity.Route;
import ua.kpi.fict.routesearch.entity.RouteSearchInputData;
import ua.kpi.fict.routesearch.service.RouteService;
import ua.kpi.fict.routesearch.service.TimingRouteSearchService;

@Service
@RequiredArgsConstructor
public class GeneticRouteSearchService extends TimingRouteSearchService {

    private final RouteService routeService;

    @Value("${default.mutation-rate}")
    private double defaultMutationRate;

    @Value("${default.is-elitism-enabled}")
    private boolean defaultIsElitismEnabled;

    @Value("${default.tournament-size}")
    private int defaultTournamentSize;

    @Value("${default.population-size}")
    private int defaultPopulationSize;

    @Value("${default.populations-in-evolution-count}")
    private int defaultPopulationsInEvolutionCount;

    @Value("${default.elite-points-count}")
    private int defaultElitePointsCount;

    private double mutationRate;
    private boolean isElitismEnabled;
    private int tournamentSize;
    private int populationSize;
    private int populationsInEvolutionCount;
    private int elitePointsCount;

    @Override
    protected Route findRoute(RouteSearchInputData inputData) {
        Population population = new Population();
        updateGeneticPropertiesIfNeeded(inputData);
```

```
        generateRandomRoutes(inputData.getPoints(), population);
        population = performEvolution(population);
        return routeService.findFittest(population.getRoutes());
    }

    private void updateGeneticPropertiesIfNeeded(RouteSearchInputData inputData) {
        if (inputData instanceof GeneticRouteSearchInputData) {
            setGeneticPropertiesFromInputData((GeneticRouteSearchInputData) inputData);
        } else {
            setDefaultGeneticProperties();
        }
    }

    private void setDefaultGeneticProperties() {
        mutationRate =  defaultMutationRate;
        tournamentSize = defaultTournamentSize;
        populationSize = defaultPopulationSize;
        isElitismEnabled = defaultIsElitismEnabled;
        populationsInEvolutionCount = defaultPopulationsInEvolutionCount;
        elitePointsCount = defaultElitePointsCount;
    }

    private void setGeneticPropertiesFromInputData(GeneticRouteSearchInputData
inputData) {
        mutationRate = isNull(inputData.getMutationRate()) ? defaultMutationRate :
inputData.getMutationRate();
        tournamentSize = isNull(inputData.getTournamentSize()) ? defaultTournamentSize
: inputData.getTournamentSize();
        populationSize = isNull(inputData.getPopulationSize()) ? defaultPopulationSize
: inputData.getPopulationSize();
        isElitismEnabled = isNull(inputData.getElitismEnabled())
            ? defaultIsElitismEnabled : inputData.getElitismEnabled();
        populationsInEvolutionCount =
isNull(inputData.getPopulationsInEvolutionCount())
            ? defaultPopulationsInEvolutionCount :
inputData.getPopulationsInEvolutionCount();
        elitePointsCount = isNull(inputData.getElitePointsCount()) ||
!defaultIsElitismEnabled
            ? defaultElitePointsCount : inputData.getElitePointsCount();
    }

    private void generateRandomRoutes(List<Point> points, Population population) {
        for (int i = 0; i < populationSize; i++) {
            Route route = Route.builder().points(new ArrayList<>(points)).build();
            Collections.shuffle(route.getPoints());
            population.getRoutes().add(route);
        }
    }

    private Population performEvolution(Population population) {
        for (int i = 0; i < populationsInEvolutionCount; i++) {
            population = evolve(population);
        }
        return population;
    }

    private Population evolve(Population population) {
        Population newPopulation = new Population(new
ArrayList<>(population.getRoutes().size()));
        int firstChangeableElementIndex = 0;
        if (isElitismEnabled) {
```

```
newPopulation.getRoutes().addAll(routeService.findSeveralFittest(population.getRoutes()
, elitePointsCount));
            firstChangeableElementIndex = elitePointsCount;
        }
        provideInheritanceInPopulation(population, newPopulation,
firstChangeableElementIndex);
        performMutationInPopulation(newPopulation, firstChangeableElementIndex);
        return newPopulation;
    }

    private void provideInheritanceInPopulation(Population population, Population
newPopulation,
        int firstChangeableElementIndex) {
        for (int i = firstChangeableElementIndex; i < population.getRoutes().size();
i++) {
            Route firstParent = selectByTournament(population);
            Route secondParent = selectByTournament(population);
            Route child = crossover(firstParent, secondParent);
            newPopulation.getRoutes().add(i, child);
        }
    }

    private void performMutationInPopulation(Population newPopulation, int
firstChangeableElementIndex) {
        for (int i = firstChangeableElementIndex; i < newPopulation.getRoutes().size();
i++) {
            mutate(newPopulation.getRoutes().get(i));
        }
    }

    private Route crossover(Route firstParent, Route secondParent) {
        Route child = initializeRoute(firstParent.getPoints().size());
        addPointsFromFirstParent(firstParent, child);
        addPointsFromSecondParent(secondParent, child);
        return child;
    }

    private Route initializeRoute(int size) {
        Route route = new Route();
        for (int i = 0; i < size; i++) {
            route.getPoints().add(null);
        }
        return route;
    }

    private void addPointsFromFirstParent(Route parent, Route child) {
        int startPos = (int) (Math.random() * parent.getPoints().size());
        int endPos = (int) (Math.random() * parent.getPoints().size());

        for (int i = 0; i < parent.getPoints().size(); i++) {
            if (i > startPos && i < endPos) {
                child.getPoints().set(i, parent.getPoints().get(i));
            } else if (startPos > endPos) {
                if (!(i < startPos && i > endPos)) {
                    child.getPoints().set(i, parent.getPoints().get(i));
                }
            }
        }
    }

    private void addPointsFromSecondParent(Route parent, Route child) {
        for (int i = 0; i < parent.getPoints().size(); i++) {
```

```
            if (!child.getPoints().contains(parent.getPoints().get(i))) {
                for (int j = 0; j < child.getPoints().size(); j++) {
                    if (child.getPoints().get(j) == null) {
                        child.getPoints().set(j, parent.getPoints().get(i));
                        break;
                    }
                }
            }
        }
    }

    private void mutate(Route route) {
        for (int i = 0; i < route.getPoints().size(); i++) {
            if (Math.random() < mutationRate) {
                int secondPointIndex = (int) (route.getPoints().size() *
Math.random());

                Point firstPoint = route.getPoints().get(i);
                Point secondPoint = route.getPoints().get(secondPointIndex);

                route.getPoints().set(secondPointIndex, firstPoint);
                route.getPoints().set(i, secondPoint);
            }
        }
    }

    private Route selectByTournament(Population population) {
        Population tournament = new Population(new ArrayList<>(tournamentSize));

        for (int i = 0; i < tournamentSize; i++) {
            int index = (int) (Math.random() * population.getRoutes().size());
            tournament.getRoutes().add(i, population.getRoutes().get(index));
        }
        return routeService.findFittest(tournament.getRoutes());
    }
}


package ua.kpi.fict.routesearch.service.impl;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import lombok.RequiredArgsConstructor;

import org.springframework.stereotype.Service;

import ua.kpi.fict.routesearch.entity.Point;
import ua.kpi.fict.routesearch.entity.Route;
import ua.kpi.fict.routesearch.entity.RouteSearchInputData;
import ua.kpi.fict.routesearch.service.PointService;
import ua.kpi.fict.routesearch.service.RouteService;
import ua.kpi.fict.routesearch.service.TimingRouteSearchService;

@Service
@RequiredArgsConstructor
public class GreedyRouteSearchService extends TimingRouteSearchService {

    private static final Random random = new Random();
    private final PointService pointService;
    private final RouteService routeService;
```

```
    @Override
    protected Route findRoute(RouteSearchInputData inputData) {
        List<Point> copiedPoints = new ArrayList<>(inputData.getPoints());
        List<Point> resultPoints = new ArrayList<>();

        Point currentPoint = chooseFirstPointAndMoveItToResult(copiedPoints,
resultPoints);
        addRemainingPointsToResult(copiedPoints, resultPoints, currentPoint);
        Route route = Route.builder().points(resultPoints).build();
        routeService.calculateDistance(route);
        return route;
    }

    private Point chooseFirstPointAndMoveItToResult(List<Point> copiedPoints,
List<Point> resultPoints) {
        Point currentPoint = getRandomPoint(copiedPoints);
        resultPoints.add(currentPoint);
        copiedPoints.remove(currentPoint);
        return currentPoint;
    }

    private Point getRandomPoint(List<Point> copiedPoints) {
        return copiedPoints.get(random.nextInt(copiedPoints.size()));
    }

    private void addRemainingPointsToResult(List<Point> copiedPoints, List<Point>
resultPoints, Point currentPoint) {
        while (!copiedPoints.isEmpty()) {
            Point nearestPoint = findNearestPointInList(currentPoint, copiedPoints);
            resultPoints.add(nearestPoint);
            currentPoint = nearestPoint;
            copiedPoints.remove(nearestPoint);
        }
    }

    private Point findNearestPointInList(Point currentPoint, List<Point> points) {
        Point nearestPoint = points.get(0);
        double nearestDistance = pointService.calculateDistance(currentPoint,
nearestPoint);
        for (int i = 1; i < points.size(); i++) {
            if (pointService.calculateDistance(currentPoint, points.get(i)) <
nearestDistance) {
                nearestDistance = pointService.calculateDistance(currentPoint,
points.get(i));
                nearestPoint = points.get(i);
            }
        }
        return nearestPoint;
    }
}


package ua.kpi.fict.routesearch.service.impl;

import org.springframework.stereotype.Service;

import ua.kpi.fict.routesearch.entity.Point;
import ua.kpi.fict.routesearch.service.PointService;

@Service
public class PointServiceImpl implements PointService {
```

```
    @Override
    public double calculateDistance(Point firstPoint, Point secondPoint) {
        int xDistance = Math.abs(firstPoint.getX() - secondPoint.getX());
        int yDistance = Math.abs(firstPoint.getY() - secondPoint.getY());
        return Math.sqrt((xDistance * xDistance) + (yDistance * yDistance));
    }
}
```

```
package ua.kpi.fict.routesearch.service.impl;

import static java.util.Objects.isNull;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import lombok.RequiredArgsConstructor;

import org.springframework.stereotype.Service;

import ua.kpi.fict.routesearch.entity.Point;
import ua.kpi.fict.routesearch.entity.Route;
import ua.kpi.fict.routesearch.service.PointService;
import ua.kpi.fict.routesearch.service.RouteService;

@Service
@RequiredArgsConstructor
public class RouteServiceImpl implements RouteService {

    private final PointService pointService;

    @Override
    public Route findFittest(List<Route> routes) {
        Route fittestRoute = routes.get(0);
        for (int i = 1; i < routes.size(); i++) {
            if (getFitness(fittestRoute) <= getFitness(routes.get(i))) {
                fittestRoute = routes.get(i);
            }
        }
        return fittestRoute;
    }

    @Override
    public List<Route> findSeveralFittest(List<Route> routes, int quantity) {
        List<Route> routesCopy = new ArrayList<>(routes);

        if (quantity > routesCopy.size()) {
            quantity = routesCopy.size();
        }
        return findNFittestRoutes(quantity, routesCopy);
    }

    private List<Route> findNFittestRoutes(int quantity, List<Route> routes) {
        List<Route> result = new ArrayList<>();

        for (int i = 0; i < quantity; i++) {
            Route fittestRoute = findFittest(routes);
            result.add(fittestRoute);
            routes.remove(fittestRoute);
        }
```

```
        return result;
    }

    @Override
    public double getFitness(Route route) {
        if (isNull(route.getFitness())) {
            route.setFitness(1 / (double) calculateDistance(route));
        }
        return route.getFitness();
    }

    @Override
    public int calculateDistance(Route route) {
        if (isNull(route.getDistance())) {
            int distance = 0;
            Iterator<Point> iterator = route.getPoints().iterator();
            Point from = iterator.next();

            while (iterator.hasNext()) {
                Point to = iterator.next();
                distance += pointService.calculateDistance(from, to);
                from = to;
            }
            route.setDistance(distance);
        }
        return route.getDistance();
    }
}
```

```
package ua.kpi.fict.routesearch.rest;

import static ua.kpi.fict.routesearch.Constants.API_BASE;
import static ua.kpi.fict.routesearch.Constants.FIND_ROUTE_BASE;
import static ua.kpi.fict.routesearch.Constants.GENETIC_ALGORITHM_BASE;
import static ua.kpi.fict.routesearch.Constants.GREEDY_ALGORITHM_BASE;

import io.swagger.annotations.ApiOperation;

import javax.validation.Valid;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import ua.kpi.fict.routesearch.entity.RouteSearchInputData;
import ua.kpi.fict.routesearch.entity.RouteSearchResult;
import ua.kpi.fict.routesearch.rest.mapper.RouteSearchDataMapper;
import ua.kpi.fict.routesearch.rest.request.GeneticRouteSearchRequest;
import ua.kpi.fict.routesearch.rest.request.GreedyRouteSearchRequest;
import ua.kpi.fict.routesearch.rest.request.RouteSearchRequest;
import ua.kpi.fict.routesearch.rest.response.RouteSearchResponse;
import ua.kpi.fict.routesearch.service.RouteSearchService;

@Slf4j
@RestController
@RequestMapping(API_BASE + FIND_ROUTE_BASE)
```

```
@RequiredArgsConstructor
public class RouteSearchResource {

    private final RouteSearchService greedyRouteSearchService;
    private final RouteSearchService geneticRouteSearchService;
    private final RouteSearchDataMapper routeSearchDataMapper;

    @GetMapping
    @ApiOperation(value = "Find a shortest route passing through all points")
    public ResponseEntity<RouteSearchResponse> findShortestRoute(
        @Valid @RequestBody RouteSearchRequest request) {
        log.info("Request to find a shortest route : {}", request);
        RouteSearchInputData inputData =
routeSearchDataMapper.toRouteSearchInputData(request);
        RouteSearchResult routeSearchResult = findShortestRoute(request, inputData);
        return
ResponseEntity.ok(routeSearchDataMapper.toRouteSearchResponse(routeSearchResult));
    }

    private RouteSearchResult findShortestRoute(RouteSearchRequest request,
RouteSearchInputData inputData) {
        if (request.isTimeCritical()) {
            return greedyRouteSearchService.findShortestRoute(inputData);
        } else {
            return geneticRouteSearchService.findShortestRoute(inputData);
        }
    }

    @GetMapping(GENETIC_ALGORITHM_BASE)
    @ApiOperation(value = "Find a shortest route passing through all points via a
genetic algorithm")
    public ResponseEntity<RouteSearchResponse> findShortestRouteViaGeneticAlgorithm(
        @Valid @RequestBody GeneticRouteSearchRequest request) {
        log.info("Request to find a shortest route via a genetic algorithm : {}",
request);
        RouteSearchInputData inputData =
routeSearchDataMapper.toGeneticRouteSearchInputData(request);
        RouteSearchResult routeSearchResult =
geneticRouteSearchService.findShortestRoute(inputData);
        return
ResponseEntity.ok(routeSearchDataMapper.toRouteSearchResponse(routeSearchResult));
    }

    @GetMapping(GREEDY_ALGORITHM_BASE)
    @ApiOperation(value = "Find a shortest route passing through all points via a
greedy algorithm")
    public ResponseEntity<RouteSearchResponse> findShortestRouteViaGreedyAlgorithm(
        @Valid @RequestBody GreedyRouteSearchRequest request) {
        log.info("Request to find a shortest route via a greedy algorithm : {}",
request);
        RouteSearchInputData inputData =
routeSearchDataMapper.toRouteSearchInputData(request);
        RouteSearchResult routeSearchResult =
greedyRouteSearchService.findShortestRoute(inputData);
        return
ResponseEntity.ok(routeSearchDataMapper.toRouteSearchResponse(routeSearchResult));
    }
}


package ua.kpi.fict.routesearch.rest.mapper;
```

```java
import org.mapstruct.Mapper;
import org.mapstruct.Mapping;

import ua.kpi.fict.routesearch.entity.GeneticRouteSearchInputData;
import ua.kpi.fict.routesearch.entity.RouteSearchInputData;
import ua.kpi.fict.routesearch.entity.RouteSearchResult;
import ua.kpi.fict.routesearch.rest.request.GeneticRouteSearchRequest;
import ua.kpi.fict.routesearch.rest.request.GreedyRouteSearchRequest;
import ua.kpi.fict.routesearch.rest.request.RouteSearchRequest;
import ua.kpi.fict.routesearch.rest.response.RouteSearchResponse;

@Mapper
public interface RouteSearchDataMapper {

    RouteSearchInputData toRouteSearchInputData(RouteSearchRequest request);

    RouteSearchInputData toRouteSearchInputData(GreedyRouteSearchRequest request);

    GeneticRouteSearchInputData toGeneticRouteSearchInputData(GeneticRouteSearchRequest
request);

    @Mapping(target = "route", source = "route.points")
    @Mapping(target = "distance", source = "route.distance")
    RouteSearchResponse toRouteSearchResponse(RouteSearchResult routeSearchResult);
}
```

```java
package ua.kpi.fict.routesearch.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;
import lombok.experimental.SuperBuilder;

@Data
@SuperBuilder
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode(callSuper = true)
public class GeneticRouteSearchInputData extends RouteSearchInputData {

    private Double mutationRate;

    private Integer tournamentSize;

    private Integer populationSize;

    private Integer populationsInEvolutionCount;

    private Boolean elitismEnabled;

    private Integer elitePointsCount;
}
```

```java
package ua.kpi.fict.routesearch.entity;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Getter;
```

```java
import lombok.NoArgsConstructor;
import lombok.Setter;
import lombok.ToString;

@Builder
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class Point {

    private Integer x;

    private Integer y;
}
```

```java
package ua.kpi.fict.routesearch.entity;

import java.util.ArrayList;
import java.util.List;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class Population {

    private List<Route> routes = new ArrayList<>();
}
```

```java
package ua.kpi.fict.routesearch.entity;

import java.util.ArrayList;
import java.util.List;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class Route {

    private List<Point> points = new ArrayList<>();

    private Double fitness;

    private Integer distance;
}
```

```java
package ua.kpi.fict.routesearch.entity;

import java.util.ArrayList;
import java.util.List;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.experimental.SuperBuilder;

@Data
@SuperBuilder
@AllArgsConstructor
@NoArgsConstructor
public class RouteSearchInputData {

    private List<Point> points = new ArrayList<>();
}
```

```java
package ua.kpi.fict.routesearch.entity;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class RouteSearchResult {

    private Route route;

    private Long millisecondsTaken;
}
```

```java
package ua.kpi.fict.routesearch.config;

import java.util.Collections;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class DocumentationConfig {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
```

```
            .apis(RequestHandlerSelectors.basePackage("ua.kpi.fict.routesearch.rest"))
            .paths(PathSelectors.any())
            .build()
            .apiInfo(getApiInfo());
    }

    private ApiInfo getApiInfo() {
        return new ApiInfo(
            "Route Search Application",
            "This is an API for finding shortest routes passing through all given
points",
            "1.0",
            null,
            null,
            null,
            null,
            Collections.emptyList()
        );
    }
}
```

```
package ua.kpi.fict.routesearch;

import lombok.experimental.UtilityClass;

@UtilityClass
public class Constants {

    public static final String VALUE_PROPERTIES_CLASSPATH =
"classpath:values.properties";

    public static final String API_BASE = "/api";

    public static final String FIND_ROUTE_BASE = "/find-route";

    public static final String GENETIC_ALGORITHM_BASE = "/genetic";

    public static final String GREEDY_ALGORITHM_BASE = "/greedy";
}
```

```
package ua.kpi.fict.routesearch;

import static ua.kpi.fict.routesearch.Constants.VALUE_PROPERTIES_CLASSPATH;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.PropertySource;

@SpringBootApplication
@PropertySource(VALUE_PROPERTIES_CLASSPATH)
public class RouteSearchApplication {

    public static void main(String[] args) {
        SpringApplication.run(RouteSearchApplication.class, args);
    }
}
```